Automatic Derivation of Dependency Chains within Systems for Automated Driving via Ontology Based Scenario Representations

Master Thesis no. 713/18

Editor: Phillip Maxim Hoßbach | 2086103

Supervisors: Christian Amersbach, M.Sc. | Dr. Michael Darms

TECHNISCHE UNIVERSITÄT DARMSTADT

FZD FAHRZEUGTECHNIK TU DARMSTADT

Phillip Maxim Hoßbach
Matriculation no.: 2086103
Study program: Master Mechatronik

Master Thesis no. 713/18
Topic: Automatic Derivation of Dependency Chains within Systems for Automated Driving via Ontology Based Scenario Representations

Submitted: 15 April 2019

Technische Universität Darmstadt
Fachgebiet Fahrzeugtechnik
Prof. Dr. rer. nat. Hermann Winner
Otto-Berndt-Straße 2
64287 Darmstadt

Thesis Statement

Thesis Statement pursuant to § 22 paragraph 7 and § 23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Phillip Maxim Hoßbach, have written the submitted thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.
I am aware, that in case of an attempt at deception based on plagiarism (§ 38 paragraph 2 of APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.
In the submitted thesis the written copies and the electronic version for archiving are pursuant to § 23 paragraph 7 of APB identical in content.
For a thesis of the Department of Architecture, the submitted electronic version corresponds to the presented model and the submitted architectural plans.

Darmstadt, 15 April 2019

Abstract

In the development of automated vehicles, the complexity of the underlying, safety-critical systems poses a challenge for today's engineers. Furthermore, mandatory standards such as ISO 26262 require traceability and assignability of elaborated requirements. In addition, due to the increasing use of agile product development processes, it is necessary to be able to deal with continuous system changes. In this work an approach is presented, which on the one hand aims at supporting design decisions of developers by representing dependencies in a system. On the other hand, an improved documentation and traceability of system requirements is intended. The proposed approach is based on the automatic derivation of so-called dependency chains. For this, ontology based representations of driving scenarios and a system's architecture modeled by directed acyclic graphs are utilized to enable associations between scenarios and particular system components.

After an introduction to the corresponding state of the art, the fundamentals of ontology engineering and the representation of system architectures are outlined. In addition a consistent terminology for driving scenarios is adopted and the situation awareness and information acquisition within driving tasks is described. Subsequently, the conceptual basis of the approach is presented. Based on the established terms a meta-model is developed, from which three key challenges concerning the intended solution are derived. These challenges are then addressed by the design of partial solutions. Not only the development methodology of the respective ontology is discussed. Also, a dedicated modeling possibility for a system's architecture, the task-chain-pattern skill graph representation, is elaborated in this context. Moreover, the construct of a chain derivation engine, which constitutes the core concept of this work, is explained. Semantic rules contained in this engine, together with arithmetic functions attached to it, enable the eventual derivation of the intended dependency chains. In order to provide a proof of concept, the developed solution proposals are implemented first separately then collectively. Therein, driving scenarios are narrowed to two elementary scenarios in order to limit the scope of the implemented ontology. Regarding the exemplary system architecture representation, one particular skill is elaborated. Therefore, dedicated examples are provided for the different components of the implementation. In this way, the process is examined in more detail. The examples are then assembled and observed collectively. This illustrates the holistic chain derivation process and achieved analysis functionalities. The implemented software framework is charged with different quantities of variously complex input information and the resulting runtimes of the chain derivation process are interpreted. Hence, its potential and limits for a utilization in the development of fully automated vehicles is evaluated.

Table of Contents

Symbols and Indices

Latin Symbols:

Symbol	Unit	Description
D_{min}	m	Minimum braking distance
OS	-	Operational scope
R	m	Range
RC	-	Rule catalog
SA	-	System architecture
d_{max}	$\frac{\text{m}}{\text{s}^2}$	Maximum deceleration
n_{r}	-	Rule count
n_{s}	-	Scenario count
r	-	Rule
r_i	-	Rule i
s	-	Scenario
s_i	-	Scenario i
v	$\frac{\text{m}}{\text{s}}$	Longitudinal velocity

Greek Symbols:

Symbol	Unit	Description
Λ	-	Complexity
α_{H}	°	Horizontal field of view
$\overline{\Lambda}$	-	Average complexity

List of Abbreviations

API	Application Programming Interface
E/E	electric-electronic
EN	Norme Européenne (fr.)
FZD	Fachgebiet Fahrzeugtechnik Darmstadt (ger.)
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
OEM	Original Equipment Manufacturer
OWL	Web Ontology Language
RDF	Resource Description Framework
RDF(S)	Resource Description Framework Schema
SAE	Society of Automotive Engineers
SPARQL	SPARQL Protocol and RDF Query Language
SWRL	Semantic Web Rule Language
UML	Unified Modeling Language
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
XML	Extensible Markup Language

List of Figures

List of Tables

1 Introduction

In order to give an introduction to the topic, this chapter starts with an outline of the motivation for research in the area of autonomous vehicle system design. The content of this work is then distinguished from existing literature by a description and further clarification of the assigned task. Eventually, based on the task, the methodology of the approach and the thesis structure are explained.

1.1 Motivation

The vision of autonomous driving is becoming more tangible every day. Even though estimates for the introduction of such a technology range from 3 to 30 years, OEMs, suppliers and software companies are already engaged in an wheel-to-wheel race and are working on their first versions of fully automated vehicles (SAE level 5).[1] In order for Germany to further consolidate its position as a pioneer in the field of automated vehicle technologies, the Federal Ministry of Transport and Digital Infrastructure has initiated a distinct funding landscape for German institutions and companies.[2] However, there are still major challenges to be addressed. These challenges not only stem from the extraordinarily high complexity of the developed systems, but are also primarily due to the fact that these systems involve many safety-critical components and sub-systems. Since fully autonomous vehicles are supposed to be able to move independently on public roads in the long term, it must be ensured that safe operation is guaranteed, i.e. that no damage is caused and no people are injured.[3]

In this context, the functional safety requirements of electric, electronic and programmable electronic (E/E) systems and components are considered. The cross-industry standard IEC/EN 61508, which formulates requirements for the functional safety of E/E-systems in general, was utilized to derive the ISO 26262 standard. It comprises definitions and guidelines for the development and testing of systems in regard to the automotive industry, as well as their documentation. In today's development of fully automated vehicles, it is to be ensured that all work products involved in the process are designed, described and tested in accordance with this standard. This means that requirements are not only demanded of the final product and its validation, but also of the development and design process that precedes it.[4]

Eventually, engineers are confronted with these requirements. This is why efforts are being made to develop process solutions that support the development process of the overall system in order to facilitate the assignability of requirements and traceability of associated design decisions. Additionally, agile development methods are commonly practiced in the development of automated vehicle systems. These require the ability to deal with continuous system changes or varying requirements on components. In conclusion, the key motivation of this work is to support the complex development process of autonomous vehicles and its conformity to corresponding standards.

[1] Matthaei, R.; Maurer, M.: Autonomous Driving – a Top-Down-Approach (2015), p. 155.
[2] Steimle, M. et al.: Terminologie für den szenarienbasierten Testansatz (2018), pp. 1-2.
[3] National Highway Traffic Safety Administration (NHTSA): Automated Driving – A Vision for Safety (2017).
[4] Menzel, T. et al.: Scenarios for Automated Vehicles (2018), pp. 1821-1822.

1.2 Description and Clarification of the Task

The aim of this work is to investigate the extent to which ontology based scenario representations may be utilized during the design and development process of a system. By means of these specific scenario representations, associations with the components of a system's architecture shall be made possible in order to automatically derive or represent dependencies in a system.

For this purpose, an initial investigation of the current state of the art is carried out. Based on this, the key topics in regard to this work are extracted and further studied. This is intended to enable the development of a tailored solution concept. The development of this concept is thereby aligned to already established terms and available tools in order to maintain consistency with the state of the art. Subsequently, by application of the concept, exemplary requirements on architectural components, e.g. sensors or actuators, are derived. Here, the semantic expressiveness of the ontology based scenario representations is exploited. Particularly, the concept describes a method which facilitates the derivation of so-called dependency chains.

The whole course of this work depends on the ontology conceived at the beginning. The ontology is developed in consideration of existing terminologies and by applying a concrete methodology. Thereby simplifications are assumed to allow a limitation of the work's scope. The ontology is implemented with a tool called *Protégé* and exemplary knowledge bases are generated manually. Thereby, the tasks differs from approaches which focus on the automatic generation of scenarios.

Finally, a software implementation of the approach is carried out in order to provide a proof of concept. All aspects of the approach are implemented using simple examples. These are then considered collectively. A presentation of the automatic dependency chain derivation and the designed analysis functionalities results. The approach developed in this work is interpreted as a kind of conceptual prototype and does not claim to be universally applicable. Both the developed ontology and the presented chain derivation process are limited to the examples modeled in this paper.

1.3 Methodology and Thesis Structure

In order to solve the tasks of this work, a particular methodology is chosen. The methodology is dividable into four phases: research, concept development, implementation and evaluation. Each of these phases is based on a methodical concept in order to facilitate appropriate actions and thus support a structured overall work flow.

Initially, various research techniques are used to obtain an overview of the current state of the art and the relevant fundamentals are acquired. The starting point for the research process is the task, or more precisely the requirements formulated therein, the correlations mentioned and the referenced sources. Thematically related publications of the Institute of Control Engineering in Braunschweig, Germany receive particular consideration, since the work of the employees at this institute (especially Bagschik et al.[5]) is strongly associated with the topic of this work. Two basic strategies according

[5] Bagschik, G. et al.: Ontology based Scene Creation for Automated Vehicles (2018).

to Rossig[6] are pursued and combined in the procurement of the literature. On the one hand, this involves starting with the most up-to-date specialist literature possible and systematically advancing to earlier, more fundamental literature with the aid of various search tools (bibliographic method[6]). On the other hand, the literature that appears relevant in already collected sources is also followed up. At first, the number of literature references rises rapidly, until finally the already known sources are found (method of concentric circles[6]). The goal is to identify the most cited, i.e. probably the most significant sources. In addition, the integrity of the literature catalog is checked and ensured by a consultation with experts, in this case the supervisors of this thesis, employees of the work group and authors of related publications. To provide an overview of key concepts, common terminologies and related work, the most significant insights from research are explained in Chapter 2.1.

Based on the previously extracted literature, a concept for the fulfillment of the task is developed. This procedure is covered by Chapter 3. The concept is based on established terminologies in order not to redefine structures that have already been proven to be conclusive. If existing terminologies are insufficient for the description of certain contexts, proposals for the extension of the definitions or a combination of different terminologies are made. With regard to an implementation, suitable languages, schemata and interfaces are selected. Particularly in the context of the ontology to be developed, decisions regarding the general setup and methodology are made on the basis of dedicated technical literature. The related elaboration is to be found in Section 3.2. In addition, the developed solution concept is discussed with experts at an early stage in order to ensure its consistency and applicability. Here, the same experts are interviewed as in the course of the literature research.

In order to be able to statue a proof of concept, the developed concept is implemented by means of examples. Thereby, a specific scenario subset serves as a foundation for the ontology to be developed. An exemplary system architecture model is also implemented in relation to the scenarios in order to be able to derive dedicated dependency chains. Since the development of ontologies necessarily has to be carried out iteratively[7], it is decided to organize the entire implementation process in an iterative manner. Consequently, the process initially launches with a simple but expandable version of the ontology and system architecture. The individual parts are then adapted and extended on the basis of requirement checks and the elaboration of necessary modifications in continuous repetitions. In the course of this process, adaptation of the overall concept developed beforehand is not excluded, since the iterations may result in general need for change. After a defined number of repetitions, the resulting work products are used to realize system analysis and dependency chain visualization functionalities. The final work products of the implementation are presented in Chapter 4.

Eventually, an evaluation of the proposed approach is carried out. Thereby, a discussion is being conducted as to whether the approach is load scalable beyond the examined examples. The implemented analysis functionalities are instrumentalized in order to be able to provide quantified conclusions on defined criteria. The results of the evaluation are to be found in Chapter 5. Finally, an assessment of the potentials and limits of the approach is conducted.

[6] Rossig, W. E.: Wissenschaftliche Arbeiten (2011).
[7] Noy, N. F.; McGuinness, D.: Ontology Development 101 (2001).

2 Fundamentals – State of the Art and Terminology

This chapter comprises fundamental basics and annotations on contributions related to this work's approach. Initially, state of the art publications which deal with corresponding issues are recapitulated. Subsequently, key aspects from the field of Ontological Engineering are presented. Afterwards, terms for the description of driving scenarios, scenes and other concepts are introduced and allocated in order to provide a consistent terminology. Next, a proposal for the representation of system architectures, which is of particular importance for this work, is explained. The chapter is then concluded by an excursion on the topic of Situation Awareness and Information Acquisition within driving tasks.

2.1 State of the Art

This work's approach addresses the automatic derivation of dependency chains within systems for automated driving. Thereby, particular circumstances in ontology based knowledge representations of driving scenarios are meant to be linked with related system components to express sub-system requirements, identify conflicts or disclose effects of changes. The aim is not only to generally support development engineers in the concept phase but particularly to facilitate the handling of continuous changes while guaranteeing process traceability at the same time.

In software engineering, the utilization of ontology based domain knowledge for the elicitation of requirements is a popular concept. Kaiya and Saeki[8] state that domain knowledge is a crucial factor in requirements engineering resp. for the derivation of high quality software requirements. They propose the ORE method (*Ontology based Requirements Elicitation*), where semantic information is used to support developers. By applying ORE, analysts are supposed to be assisted in issues like whether further requirements should be added for improving completeness of a current software or which requirements should be discarded for keeping consistency. They deal with requirement descriptions in natural language and finally evaluate their method by an experimental case study of a music player software. Lasheras et al.[9] present a related approach. They also address an ontology based requirements engineering issue, yet rather focus on security requirements based on risk analysis methods. Eventually, a framework for representing, storing as well as reusing security requirements is suggested. Siegemund et al.[10] also contribute research on ontology-driven requirements engineering. They confirm the sufficient expressiveness of ontologies in regard of capturing requirements. Furthermore, it is stated, that by using ontologies, reasoning tasks can be performed to review the consistency and completeness of requirements. On the one hand these publications deal with both ontologies and requirements, however they all focus more on the representation of requirements instead of their derivation.

[8] Kaiya, H.; Saeki, M.: Using Domain Ontology for Requirements Elicitation (2006).
[9] Lasheras, J. et al.: Security Requirements Based on an Ontology (2009).
[10] Siegemund, K. et al.: Ontology-Driven Requirements Engineering (2011).

Mayer et al.[11] as well as Maier et al.[12] suggest the use of ontologies to optimize design-driven development processes. They introduce a framework which exploits artifact and simulation models to integrate multidisciplinary design optimization processes. Their aim is to overcome representational and semantic differences between analysis disciplines and execution environments. Mainly, high-level process constraints and development meta models are addressed. Therefore, even if the eventual design process is a key element of their approach, only a few parallels, namely the utilization of an ontology and design support ambitions, might be drawn to this work's approach.

Another subject, which is central for this work, is the representation of system architectures, especially the depiction of sub-system information and methods for tracing back to individual system components. In this context, the approach of Hebisch et al.[13] is worth mentioning. They aim to enable the evaluation of design decisions even before an architecture exists. This is achieved by creating architectural alternatives directly from requirements and the use of so-called architecture trace diagrams (ATDs) to model the interplay of components involved in the execution of a process. In contrast to this work's approach, the focus is mainly on detecting design issues in regard to overall architecture models and not to address requirements on certain architecture components explicitly. Another suitable approach concerning system architecture representations is presented by Nolte et al.[14]. They suggest the use of ability and skill graphs to provide a basis for the development process of automated road vehicles by modeling systems at a particular functional level. Also, a representation method of a system's or its components' performance level is proposed. Their approach eventually leads to a monitoring and self-awareness concept for automated vehicles, which is in contrast to this work. However, the idea of a linking from functional descriptions to sub-system performance levels is considered as the latest state of the art and adapted in this work's approach.

Ultimately, the ontology based representation of driving scenarios is a part of this work. Research of the Institute of Control Engineering at TU Braunschweig, more precisely the Vehicle Electronics Department incorporated therein, broadly addressed this topic in recent years. Bagschik et al.[15] state that requirement specifications can be encouraged by systematically identified scenario catalogs in the development process of automated vehicles. In this regard, they review ontologies as supporting knowledge based systems. They automatically generate a number of scenes in natural language to define and investigate the safe behavior of an automated vehicle or to derive test cases for simulation environments. Although this work's approach is concerned with the utilization of ontology based scenarios rather than their creation, the suggested way in which scenarios are represented is taken into account for this work.

[11] Mayer, W. et al.: Ontologies for Design-Driven Development Processes (2008).
[12] Maier, F. et al.: Ontology-Based Design Optimisation Support (2008).
[13] Hebisch, E. et al.: Architecture Trace Diagrams (2015).
[14] Nolte, M. et al.: Towards a Skill- and Ability-Based Development Process (2017).
[15] Bagschik, G. et al.: Ontology based Scene Creation for Automated Vehicles (2018).

2.2 Ontological Engineering

Following, fundamental definitions in regard to ontologies and Ontological Engineering, which are necessary to gain an understanding of subsequent chapters, are explained. First, the term ontology is examined in more detail and its definition for this work is clarified. Furthermore, well-established ontology schemata, languages, development methodologies and tools are outlined.

2.2.1 What is an Ontology?

The investigated literature encompasses a variety of definitions about what an ontology is and also addresses how such definitions have evolved over time. Originally, the term ontology refers to a philosophical discipline, which deals with the nature and structure of reality resp. with the study of attributes that belong to things because of their very own nature.[16] Computer scientists and knowledge engineers have adopted the term to denote computational artifacts which formally specify a common understanding of some domain's structure, i.e. its concepts and relations. In this regard, Neches et al.[17], Gruber[18] and Borst[19] each proposed a popular, yet unique definition of the term in the 1990s, some of which were partly related to each other.[20] Studer et al.[21] thereupon examined and merged the offered notions, hence stated that

> *"An ontology is a formal, explicit specification of a shared conceptualization."*

and added the following declarations to their definition:

> *"Conceptualization refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their use are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared reflects the notion that an ontology captures consensual knowledge, that is, it is not private of some individual, but accepted by a group."*

The latter definition by Studer et al.[21] is well-established, often applied or even referred to as "*the best suitable one*" [22] in the field of computer science nowadays.[20] Furthermore, the term ontology is distinguished from the terms glossary and taxonomy. Here, glossaries only comprise a simple set of definitions about certain terms, i.e. concepts. Taxonomies further classify these concepts in a hierarchical structure, but only constitute the backbone of ontologies, which additionally contain various properties of concepts and are semantically enhanced by particularly expressing relations between them. By extending the characteristics of glossaries and taxonomies, ontologies therefore represent

[16] Guarino, N. et al.: What is an Ontology? (2009), p. 1.
[17] Neches, R. et al.: Enabling Technology for Knowledge Sharing (1991), p. 40.
[18] Gruber, T. R.: A Translation Approach to Portable Ontology Specification (1993), p. 199.
[19] Borst, W. N.: Diss., Construction of Engineering Ontologies (1997), p. 12.
[20] Gómez-Pérez, A. et al.: Ontological Engineering (2004), p. 6.
[21] Studer, R. et al.: Knowledge Engineering: Principles and Methods (1998), p. 185.
[22] Bermejo, J.: A Simplified Guide to Create an Ontology (2007), p. 2.

the most complex form in this regard.[23] Computational ontologies, as stated by most definitions, also enclose only a certain domain of interest. In other words, solely concepts and relations which are useful to a developer's intended purpose are relevant and included in an ontology.[24]

The area of Ontological Engineering offers different expressions for certain ontology components and the related terms are not used uniformly. In the further course of this work, to promote consistency, domain concepts are constantly referred to as *classes*. For describing these classes in more detail, *properties* are used and *relations* depict connections between distinct classes. Besides that, it is possible to create instances of classes, subsequently referred to as *individuals*. Together with a set of such individuals, an ontology evolves and embodies a so-called *knowledge base*.[25] To model the domain of interest in an even deeper way and provide further restrictions on domain semantics, additional statements can be included in the ontology.[26a] In particular cases these statements are called rules, yet in this work they are referred to as *axioms*.

Another interesting aspect when working with ontologies is inference, occasionally also described as reasoning. Inference, more precisely an inference engine, allows to draw semantic conclusions about previously undefined class properties and relations from given facts by utilizing supplemental information. Such supplemental information might be expressed both by so-called inference rules, which are typically built-in concepts of ontology frameworks, and axioms, which are included additionally as explained earlier. An exemplary inference rule, besides many more, is the transitivity rule. It states that if an individual X has a relation R to an individual Y and Y has the same relation R to an individual Z, then R also consists between X and Z. Such rules and analogous axioms enable extensive conclusions from given circumstances and add major value to the semantics of an ontology resp. a knowledge base.[27]

2.2.2 Ontology Schemata and Languages

As stated in Section 2.2.1, an ontology in the context of computer science is meant to be modeled not only explicitly but formally. Therefore, computer-processable standards with well-defined syntaxes and preferably high flexibility and extensibility capabilities are required. The objective of creating such standards, especially in regard to semantic web applications, is what the *World Wide Web Consortium*[28] (W3C) has committed to. In recent years, working groups affiliated with the consortium created fundamental standards as *RDF*, *RDF(S)* and *OWL* which are well-established in the knowledge engineering community nowadays and outlined hereafter.[29a]

[23] Karbe, T. et al.: State of the Art for Automotive Ontology (2014), p. 10.
[24] Guarino, N. et al.: What is an Ontology? (2009), p. 2.
[25] Noy, N. F.; McGuinness, D.: Ontology Development 101 (2001), p. 3.
[26] Gómez-Pérez, A. et al.: Ontological Engineering (2004), a: p. 8.
[27] Busse, J. et al.: Actually, What Does "Ontology" Mean? (2015), p. 32.
[28] W3C: World Wide Web Consortium (2019).
[29] Hitzler, P. et al.: Semantic Web (2008), a: p. 11.

The *Resource Description Framework*[30] (RDF) was originally introduced as a language for representing information of web resources in a generalized manner. Thus, not only to represent meta data about such resources but also information about certain web-identifiable things. Examples for such web-identifiable things are items available on shopping websites, including their related features or lodging entities on accommodation booking websites.[30] Later on, RDF was then also defined

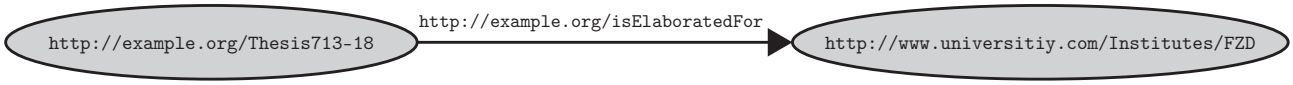


Figure 2-1: Simple RDF graph example to describe properties of classes and individuals and relations between them. The figure exemplifies that there is a certain relationship between this thesis and the corresponding institute for which it is being elaborated.[31a]

as a core format on which more complex languages for the representation of semantic information were to be built. Eventually, RDF is rather understood as a data representation model than as an actual language.[31b] In RDF resources resp. things are identified by using *Uniform Resource Identifiers* (URIs) and described in terms of simple relations and property values. This makes it possible to represent appropriate data as graphs of nodes and edges, in which both nodes and edges are uniquely labeled with URIs.[30] Figure 2-1 illustrates an example of such a graph with two nodes and one connecting edge. Then again, in information technology exist several common ways to express such graphs in a computer-processable fashion. Because RDF graphs are generally rather light, which means that only a few of all imaginable connections apply, it is advisable to represent a graph in this context as a set of effectively existing edges.[31b] This is achieved by a notation with so-called *triples*. In this notation, each statement of a graph is written as a simple concatenation of a subject, a predicate and an object.[30] For example, in the triples notation the graph shown in Figure 2-1 would be expressed by linking the three URIs together:

```
http://example.org/Thesis713-18  http://example.org/isWrittenFor
http://www.university.com/Departments/FZD .
```

RDF is therefore used to make fundamental statements about relationships between individual objects. To be able to further specify terminological knowledge in the form of class and property hierarchies and their interrelations, RDF is extended by *RDF Schema*[32] (RDF(S)).[31c] RDF(S) provides a data-modeling vocabulary with whose terms such specifications are facilitated in RDF data.[32] The RDF(S) vocabulary, however, is not a thematic vocabulary in which new terms for a specific domain are offered. Rather, the philosophy of RDF(S) is to provide universal expression means that enable statements about the semantic relationships of concepts of any user-defined vocabulary.[31c] The ability to specify schema knowledge ultimately makes RDF(S) a knowledge representation resp. ontology language with which it is possible to describe a whole range of semantic dependencies oc-

[30] McBride, B. et al.: RDF Primer (2004).
[31] Hitzler, P. et al.: Semantic Web (2008), a: based on Fig. 3.1, p. 36; b: p. 40; c: p. 67.
[32] McBride, B. et al.: RDF Schema 1.1 (2014).

curring in any domain.[33a] For example, the RDF(S) vocabulary provides a predefined possibility to explicitly specify subclass relationships between classes using the expression `rdfs:subClassOf`.[33b]

The expressiveness of RDF(S) is limited to a given extent though and only allows modeling of simple ontologies.[33c] Semantic statements like "Each project has at least one employee." or "The superior of my superior is also my superior." for example cannot be expressed in RDF or RDF(S). In order to represent such complex knowledge, even more expressive languages are required. The W3C standardized the ontology language *Web Ontology Language*[34] (OWL) for this purpose in 2004. OWL has enjoyed steadily increasing acceptance in a wide range of applications since then and is well-established in the field of ontology engineering nowadays.[33c] It is explicitly "*designed for the use by applications that need to process the content of information instead of just presenting information to humans*".[34] An OWL ontology essentially consists of classes (nodes) and properties (edges), similar to RDF(S). However, in OWL these classes and properties can be set in much more advanced relationships, i.e. cardinality, disjointness or equality properties and others are supported.[34] Higher expressiveness of a language is yet usually accompanied by a high degree of complexity. Therefore, differently expressive accents of the language were designed by the W3C. Users are able to choose between OWL Full, OWL DL and OWL Lite in accordance to the requirements of their ontology.[33c] For a detailed explanation of the specific advantages and disadvantages of the three sub-languages, readers are referred to Section 4.2 of "Foundations of Semantic Web Technologies" by Hitzler et al.[35].

Yet, as a matter of fact, the expressiveness of OWL is limited, too. A prominent example for a class relation which is not feasible in OWL, is the "Uncle-Nephew-Relationship". With the semantic terms provided by OWL it is just not possible to express the relation of a person A being the parent of a person B with A having a brother C, thus C being the uncle of B.[36] If one intends to formulate even more complex axioms to specify the knowledge about a given domain, OWL obviously becomes insufficient. For this reason the capabilities of OWL, more specifically OWL DL and OWL Lite, were combined with a sub-language of the *Rule Markup Language*[37] (RuleML) and eventually the *Semantic Web Rule Language*[36] (SWRL) was proposed. SWRL is meant not only to provide the tools for composing more complex semantic axioms but also to simplify the way of how such statements are formulated in the first place. Here, axioms are verbalized in an abstract high-level syntax as implications between an antecedent and a consequent.[36] Using this syntax, an axiom asserting the aforementioned "Uncle-Nephew" example would be formulated as:

```
isParentOf(?A, ?B) ^ isBrotherOf(?A, ?C) -> isUncleOf(?C, ?B).
```

Thus, there is a number of ways to specify information, more precisely knowledge about a specific domain, in a formal machine-readable way. Ontologies and knowledge bases modeled by the variously complex schemata and languages then serve as the foundation of different practical appli-

[33] Hitzler, P. et al.: Semantic Web (2008), a: p. 67; b: p. 71; c: pp. 125-126.
[34] Patel-Schneider, P. F. et al.: OWL Web Ontology Language (2004).
[35] Hitzler, P. et al.: Foundations of Semantic Web Technologies (2010), p. 40.
[36] Horrocks, I. et al.: SWRL: A Semantic Web Rule Language (2004).
[37] The Rule Markup Initiative: RuleML – Realize your Knowledge (2019).

cations. However, in order to actually use the specified knowledge, it is beneficial to be able to query contained information in a similar way as it is known from databases. The most common solution to do so is offered by the graph based *SPARQL Protocol and RDF Query Language*[38] (SPARQL).[39] According to E. Prud'hommeaux et al.[38] "*SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or [...] via middleware*". At its core, SPARQL is based on simple queries in a RDF-like syntax, but it also includes advanced functions for constructing more complex query patterns, for filter routines and outcome formatting.[39] As an example, a basic SPARQL query related to the RDF graph in Figure 2-1 may be formulated as follows:

```
SELECT ?thesis
WHERE
{
   ?thesis <http://example.org/isElaboratedFor>
           <http://www.university.com/Institutes/FZD> .
}
```

On the one hand, as shown, a query consists of a `SELECT` clause, which identifies the variables to appear in the expected results, and on the other hand of a `WHERE` clause, which provides the basic graph pattern to match against the respective knowledge base data. When formulated correctly, the result of the exemplary query would be → `http://example.org/Thesis713-18`.

RDF(S) as well as OWL, SWRL and SPARQL are part of this work's approach. Their relevance and utilization are outlined in Chapter 3 and Chapter 4.

2.2.3 Ontology Development

Typically, when working with ontologies, the developed ontology as a digital artifact is not the goal in itself. Much rather, the eventual use of the ontology within a particular application or the like is to be understood as the final aim.[40a] Concerning this work for example, an ontology for the representation of automotive scenarios is developed. The goal, however, is to use it for the derivation of dependency chains in a greater context. Accordingly, it is often necessary to take a step back and look at the big picture when developing an ontology. In addition, ontology development is characterized by a major creative component.[41] Therefore, several methodologies and design guidelines have been elaborated to tackle some of these issues. To give an overview instead of discussing them explicitly, key similarities of some established methodologies are outlined in the following.

It is commonly acknowledged that "*there is no single correct ontology-design methodology*"[40b], which is mostly justified by the wide range of application and thus substantial differences in the

[38] Prud'hommeaux, E.; Seaborne, A.: SPARQL Query Language for RDF (2008).
[39] Hitzler, P. et al.: Semantic Web (2008), p. 202.
[40] Noy, N. F.; McGuinness, D.: Ontology Development 101 (2001), a: p. 2; b: p. 3.
[41] Mizoguchi, R.: Ontology Development, Tools and Languages (2004), p. 62.

requirements for an ontology. Nevertheless, different ontology development methodologies repeatedly suggest the following high-level activities and ask similar questions [42a] [43]:

- **Pre-Development Process**
  This step not only includes a general feasibility study but also the identification of the problem and opportunity area as well as the selection of the most promising target area resp. domain. Asked questions range from "*Is it possible to build the ontology?*" to "*In which application will the ontology be integrated?*".
- **Specification**
  Here, the developer specifies fixed prerequisites to create guardrails for the subsequent steps. The specification might state why exactly the ontology is being built, e.g. including what its intended use is and who the final users are.
- **Conceptualization**
  The outcomes of the conceptualization step are meant to be well-structured and meaningful models related to the domain knowledge at a rather abstract level, whereas the question "*How are the different domain aspects, theories and techniques best integrated?*" is a key issue.
- **Formalization**
  In this step, as the term implies, the previously elaborated concept is transformed into a formal or already semi-computable model.
- **Implementation**
  By implementing the formalized knowledge in course of this step, a specific ontology schema or language is utilized to gain a computable model resp. the computational ontology. At this point, the developer typically commits to not only a language but also a suitable ontology development tool.
- **Post-Development Process**
  "*Who will maintain the ontology?*", more specifically "*Who will update and correct the ontology if needed?*" and "*Are others interested in (re)using the ontology?*" are typical questions asked after the core development steps have been performed.

In addition to the eminent but mostly quite specific and theoretical methodology proposals, there are also publications with a greater practical focus. Noy and McGuinness[44a] for instance offer a guide, which does not only take several of the most popular methodologies into account but also specifically builds on the practical experience of the authors. Besides, their guide particularly addresses beginners which are new in the field of ontology development. As further outlined in Chapter 3.2.1, the methodology of Noy and McGuinness[44b] is applied in this work.

[42] Gómez-Pérez, A. et al.: Ontological Engineering (2004), a: pp. 109-110, p. 151.
[43] Bermejo-Alonso, J. et al.: Ontology Engineering for Autonomous Systems (2013), p. 263.
[44] Noy, N. F.; McGuinness, D.: Ontology Development 101 (2001), a: –; b: pp. 4-11.

To provide help for ontology developers, several interfaces and tools have been created in the Ontological Engineering field over the years. There is a wide variety of such tools today and their areas of application range from not only *ontology development* but also *ontology evaluation*, *merging*, *querying* and *learning*.[45a]

Because of the aim of this work's approach, only tools dedicated to the development of an ontology are considered. Gómez-Pérez et al. describe the "*most representative ontology development tools*" in detail.[45b] General information about the tools and developers is offered and the applied knowledge models as well as the provided browsing, inferencing and even more advanced functionalities are investigated. They conclude with a very detailed, tabular comparison of eight, in their opinion most relevant, tools and tool suites.[45c] After all, the right choice depends on numerous factors. For this work, the decision for a certain ontology development tool mainly depended on the author's level of experience, on his intentions resp. the formulated task requirements and last but not least on recommendations of familiar researchers. For this work, the open source ontology development tool and editor *Protégé*[46] is used. *Protégé* is developed at the Stanford University and recently announced to have over 320,000 registered users.[46] Its utilization is subject of Section 4.2.

[45] Gómez-Pérez, A. et al.: Ontological Engineering (2004), a: pp. 293-294; b: pp. 299-338; c: pp. 354-360.
[46] Stanford Center for Biomedical Informatics Research: Protégé (2016).

2.3 Terminology of Driving Scenarios

The approach presented in this work is based on the representation of certain scenarios. But what exactly is understood as a *scenario* and which aspects are part of it? In order to investigate this question, the term is distinguished from other, similar terms in the following. Thereby, the establishment of a common understanding in the community is broken down via a discussion of prior publications.

In the literature, that is discussed in the following, various, to some extent contradictory, definitions for the term *scenario* are to be encountered. Over time, though, various researchers made it their task to define and establish a consistent terminology for the phrase as well as for related concepts like *situation*, *scene* and *scenery*.

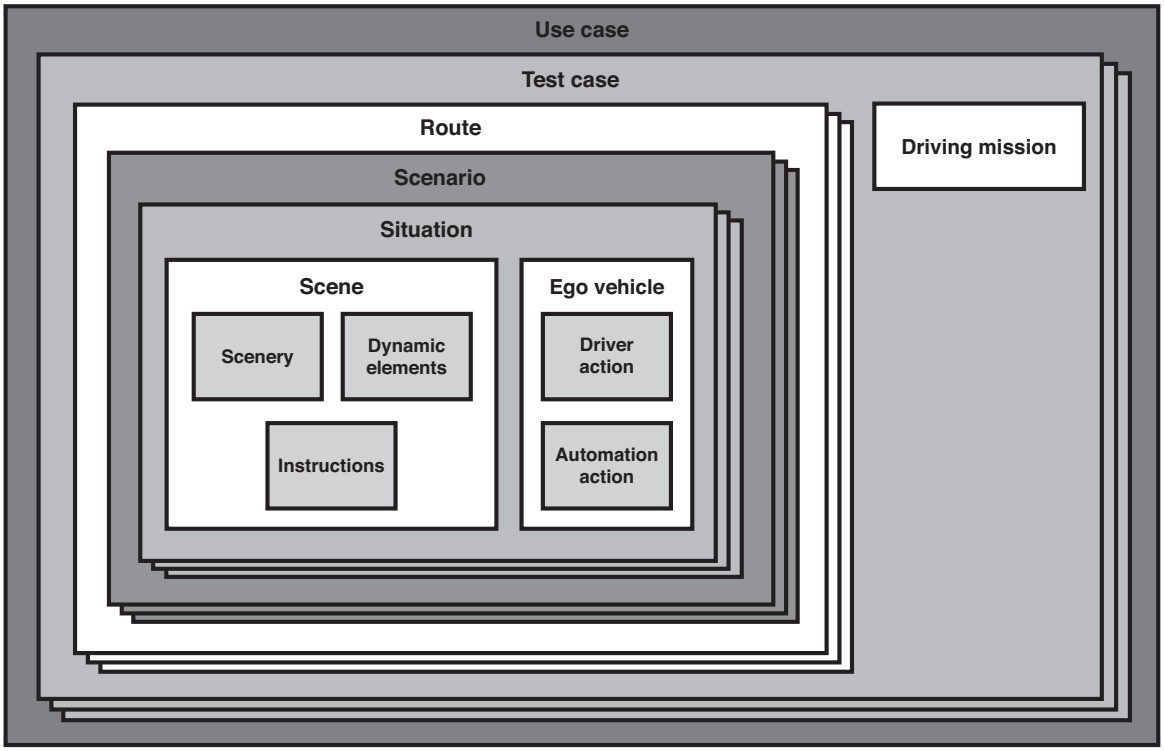


Figure 2-2: Fundamental specification of key terms for assisted and automated vehicle guidance by Geyer et al., where a scenario consists of several situations which are determined by a scene that further includes a scenery.[47a]

Geyer et al.[47b] perform a metaphor based discussion leading to a proposal for a fundamental terminology in regard to test and use-case catalogs. Their aim is mainly to facilitate the establishment of consistent terms to simplify communication and the exchange of findings within the research community.[47c] They use an analogy to the theater and movie context, i.e. to established terms in film scripts and screenplays. In doing so, not only propositions for the terms situation, scenario, scene and scenery are made but also for a route, a driving mission and an ego vehicle. As to be seen in Figure 2-2, the terms are also brought into relation with each other. Because of this, originally, the authors use the phrase ontology to label their proposal. To avoid confusions with earlier sections of this work,

[47] Geyer, S. et al.: Ontology for Test and Use-case Catalogues (2014), a: based on Fig. 1, p. 185; b: –; c: pp. 183-184.

however, the term is not adopted, instead it is referred to as a terminology in the following. In summary, it is stated that a scenario "*includes at least one situation within a scene including the scenery and dynamic elements*" and that a scenario's end "*is defined by the first irrelevant situation*".[48a]

Ulbrich et al.[49a] also observe that terms in regard to the modeling of driving contexts are not clearly defined and often used inconsistently. They take the proposal of Geyer et al. into account, yet suggest several rectifications. A clear difference for example resides in the understanding of a scene. Instead for a scene to last over a certain period of time, it is understood more as a momentary recording, namely a snapshot.[49b] Furthermore, contradictory to the concept of Geyer et al., they suggest not only to include environmental aspects into a scene but also the concept of a self-representation entity. Additional details about their interpretation of the term scene are outlined in Figure 2-3. After all, the term scenario is examined in more detail and a definition is proposed. It is considered that the scenario concept is primarily utilized to describe driver assistance systems on a functional basis.[49b] In their understanding scenarios consist of multiple linked scenes, thus, in contrast to scenes themselves, a scenario spans a certain amount of time. Also, a scenario is meant to include at least one initial scene. Thereby, the scenes included in a scenario are linked by actions and events. As shown in Figure 2-4, a scenario is here understood as one "*path of a temporal sequence of actions&events [...] and scenes*" out of "*the entirety of all possible [...] scenarios for a given initial scene*".[49c]

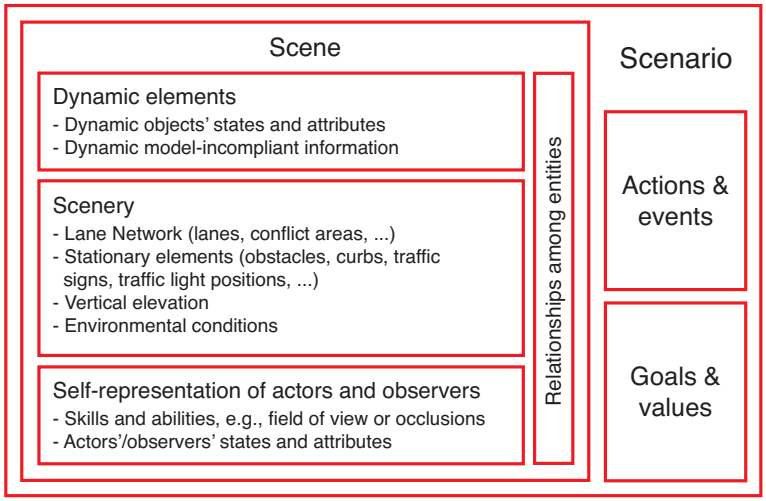


Figure 2-3: Correlation between the terms scenario and scene as well as related entities proposed by Ulbrich et al.[49a]. In contrast to Geyer et al.[48b], here, the ego vehicle resp. self-representation is part of the scene itself.[49d]

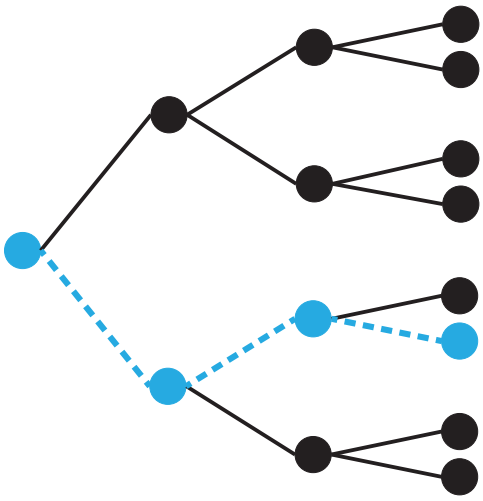


Figure 2-4: Scenario (blue dotted) interpreted as the path of a sequence of actions & events (edges) and scenes (nodes).[49e]

Menzel et al.[50] then further refine the scenario definition of their colleagues Ulbrich et al. because they discover that, depending on their eventual use, scenario representations have to focus on con-

[48] Geyer, S. et al.: Ontology for Test and Use-case Catalogues (2014), a: pp. 186-187, b: –.
[49] Ulbrich, S. et al.: The Terms Scene, Situation and Scenario (2015), a: –; b: p. 1; c: pp. 5-6; d: based on Fig. 2 and Fig. 8, pp. 2-5; e: based on Fig. 7, p. 5.
[50] Menzel, T. et al.: Scenarios for Automated Vehicles (2018).

trary aspects.[51a] The approach is based on a consideration of the ISO 26262. In that context, it is stated that different levels of detail and ways of notion are required to accomplish the work products of the related process steps. Therefore, certain levels of abstraction are identified and their necessity is examined. As a result, they propose a more granular terminology of the scenario term, still taking prior publications into account. Three levels of abstraction are suggested: *functional*, *logical* and *concrete* scenarios.[51b] As to be seen in Figure 2-5, the degree of abstraction decreases but the number of potential representations increases in the mentioned order. Functional scenarios are meant to be expressed in natural language and on a semantic level. They accordingly include a "*linguistic and consistent description of entities and relations*" [51b]. This in turn requires a consistent vocabulary, which contains terms for those entities as well as phrases for the relations between them. The authors recommend to utilize functional scenarios for certain steps during the concept phase of the ISO 26262, e.g. the item definition, hazard analysis or risk assessment. Logical scenarios, which are intended to be less abstract, are formulated on a state space level. Therefore, entities and their relations are represented with the help of parameter ranges in the state space. This format aims to cover all elements that are necessary to derive technical requirements for a system which is meant to solve the related scenario. Thus, the authors state that logical scenarios "*may be used to derive and represent requirements [...] during the system development phase*" [51b]. Concrete scenarios are, as the name implies, characterized by concrete values which are derived by selecting distinct values from given parameter ranges. Hence, every logical scenario can be migrated to a number of concrete scenarios. It is also noted that practical test cases can only be converted from such concrete scenarios. For this reason, concrete scenarios "*may be used as a basis for test case generation in the testing phase*" of the ISO 26262.[51b]

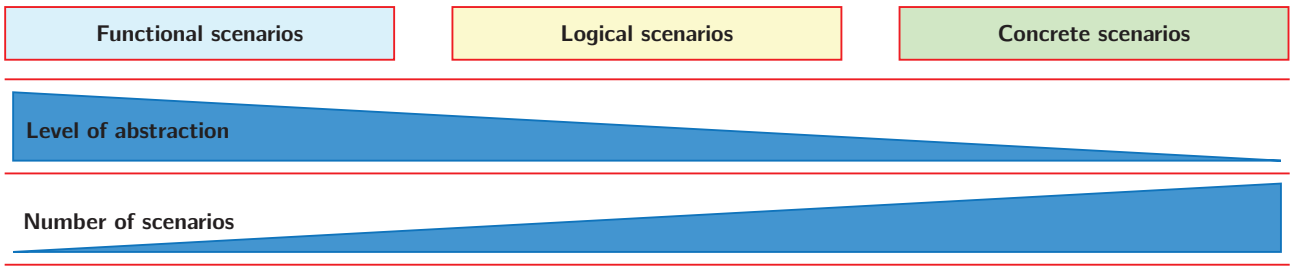


Figure 2-5: Levels of abstraction and quantity along the three scenario sub-categories suggested by Menzel et al.[51c]. Functional representations have a high degree of abstraction. The more concrete resp. less abstract a scenario is represented in turn, the more configurations are feasible.[51d]

Steimle et al.[52] also subsequently contribute to the establishment of a consistent terminology in this regard. With a focus on scenario based test approaches for automated driving functions, they extend the suggestions of Menzel et al. and propose a broad unified terminology. So far there is only a German version, but most of the key terms might be translated by observing former publications of

[51] Menzel, T. et al.: Scenarios for Automated Vehicles (2018), a: p. 1821; b: p. 1824-1826; c: –; d: based on Fig. 2, p. 1825.
[52] Steimle, M. et al.: Terminologie für den szenarienbasierten Testansatz (2018).

their colleagues in German and English. Steimle et al. suggest a depiction of the identified relevant terms and their correlations as simple UML class diagrams. This intends to resolve remaining contradictions and inaccuracies within latest terminologies by clearly visualizing both vocabulary and structure, including cardinality and aggregation information.[53a] A translated example of such a UML diagram is illustrated in Figure 2-6. On closer inspection, the exemplary diagram in fact reveals no more than an enhanced illustration of the prior proposed terminology by Ulbrich et al.[49]. Evaluating all UML diagrams in their publications however, elucidates that several further terms and relations, especially concerning scenario based testing, are declared.

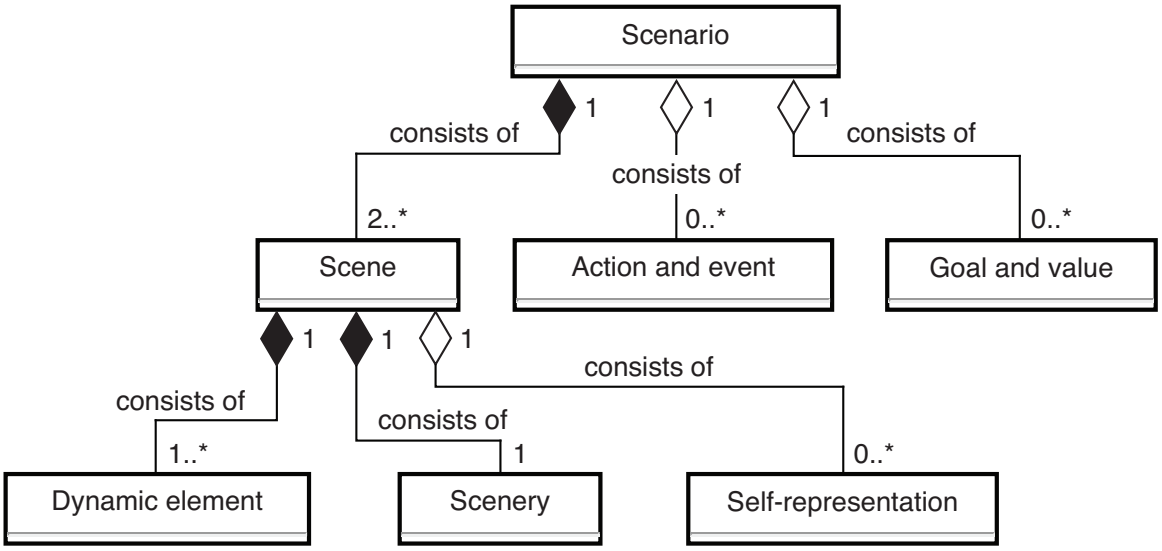


Figure 2-6: Terminology of scenarios depicted as a UML class diagram (translated from German). In contrast to prior contributions, Steimle et al.[53b] hereby additionally include cardinality and aggregation information in their illustration.[53c]

In order to comply with state-of-the-art publications and to provide consistency, this work is aligned with the previously explained and established terminologies. More precisely, the terms proposed by Ulbrich et al.[54] and the related UML diagrams by Steimle et al.[53b] are adapted. These coherences particularly reappear in Section 3.1 in which the meta model for the concept of this work's approach is outlined and in Section 4.2 as the developed ontology is described.

[53] Steimle, M. et al.: Terminologie für den szenarienbasierten Testansatz (2018), a: p. 1; b: – ; c: based on Fig. 1, p. 5.
[54] Ulbrich, S. et al.: The Terms Scene, Situation and Scenario (2015).

2.4 Representations of System Architectures

The overall goal of this work is to derive dependency chains within systems for automated driving. As stated in Section 1.2, these chains are meant to link certain circumstances in exemplary driving scenarios to concrete components of a system's architecture. It is therefore not only important to model scenarios in a particular manner but also necessary to represent the system's architecture related to the given task in some appropriate way. Following, this issue is addressed by deliberation of related state-of-the-art approaches regarding architecture representations. The quantity of publications in this field is tremendous, hence, only those that are of greater interest for this work are presented.

2.4.1 Skill and Ability Graphs

In the area of driver assistance systems and automated driving the modeling of a system in regard to its abilities and skills appears to be appropriate for several applications. Reschka[55a] and Reschka et al.[56] recapitulate modeling suggestion from 15 years of research on this matter, beginning in the early 2000s. From this, they deduce a combination of the prior approaches and present a concept for both the modeling and monitoring of vehicle guidance systems.

Initially, Reschka et al.[56] state that the terms skill and ability have been used inconsistently in the past and that significant use might be expected from deliberately defining both terms in the context of vehicular system modeling. In fact, due to a translation error, Reschka et al.[56] then confuse the terms themselves.[57a] Yet, together with their colleagues Nolte et al.[57b] they clarify their suggestions afterwards. As part of that, Nolte et al.[57a] translate the propositions of Reschka[55b] from German to English and finally offer following definitions:

- **Skill**
  A skill describes an activity of a technical system which has to be executed to fulfill the defined goals of the system.
- **Ability**
  An ability describes the quality level of an activity dependent on internal properties and the current operational situation of the system.

Apparently, there is a fine line between both terms. The distinction gets clearer as one compares the different suggested areas of application. Multiple skills are meant to collectively embody skill graphs which in turn are supposed to be utilized in the development phase of the ISO 26262 standard. In this context, a skill graph shall support not only the item definition but also the derivation of safety and technical requirements on a functional basis.[57a] Skill graphs are furthermore transformable into so-called ability graphs, which consist of several abilities and, on the contrary, might rather be applied in the operation resp. use phase of a system. Thereby, ability graphs are mainly meant to facilitate

[55] Reschka, A.: Diss., Fertigkeiten- und Fähigkeitengraphen (2017), a: – ; b: pp. 64-67.
[56] Reschka, A. et al.: Ability and Skill Graphs (2015).
[57] Nolte, M. et al.: Towards a Skill- and Ability-Based Development Process (2017), a: p. 3; b: –.

runtime monitoring and self-awareness of a system's performance.[58a] In order to practically apply an ability graph at runtime, the integration of certain performance metrics, which yield the basis for determining abilities' performance levels, is suggested. As depicted in Figure 2-7, the proposed concept is represented in the form of a directed acyclic graph. Edges constitute dependencies between abilities, whereas abilities themselves are represented as nodes.[58a] A graph of this form also includes nodes for data sources as well as data sinks and refers to a single major ability, which is represented by a superior main node. Besides, as suggested by Nolte et al., each ability node is assigned to one of the categories sense, plan or act.[58b] Additionally, through a consequent use of performance metrics, ability graphs enable the inclusion of redundancy information and are meant to allow degradation mechanisms.[59a] Reschka et al. conclude their contribution by stating that "*the main challenges for future use will be the identification of abilities and skills, their dependencies, and the necessary metrics*" [59b].

The idea of representing a system's functional architecture with skill and ability graphs is adapted in this works approach. Yet, it is slightly modified as further explained in Section 3.3.

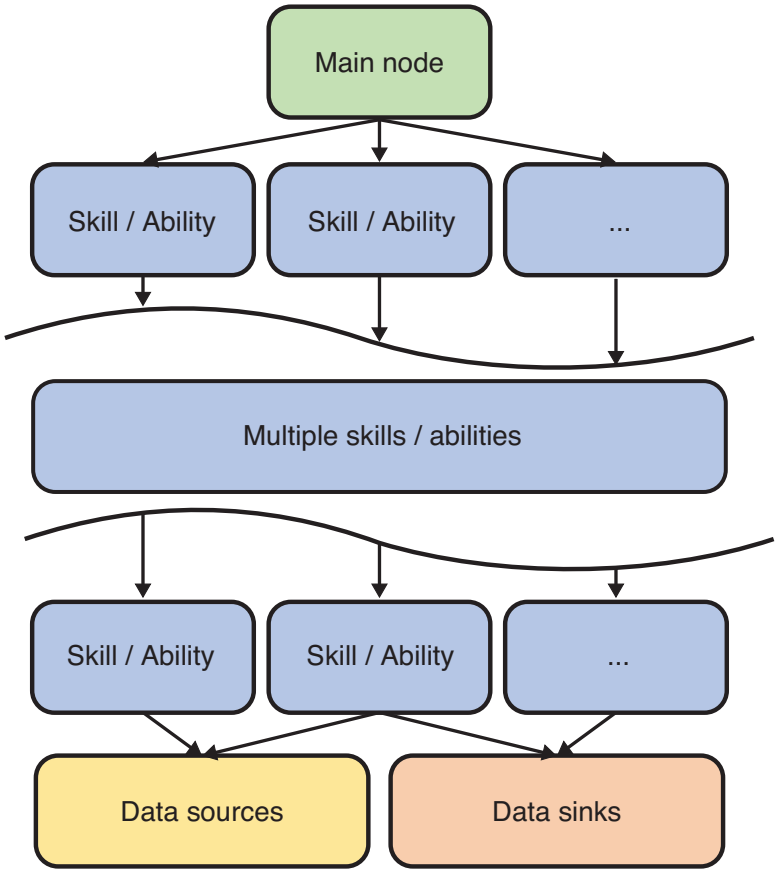


Figure 2-7: Basic skill resp. ability graph representation by Reschka et al.[59c] with arbitrarily many connected nodes. The main node represents a main ability. Data sources and sinks depict sensors and actors.[59d]

[58] Nolte, M. et al.: Towards a Skill- and Ability-Based Development Process (2017), a: p. 3; b: p. 4.
[59] Reschka, A. et al.: Ability and Skill Graphs (2015), a: p. 4; b: p. 7; c: –; d: based on Fig. 2, p. 4.

2.4.2 Viewpoints and Correspondences

Generally, various possibilities for the modeling of a system's architecture, sometimes called architecture views or viewpoints, exist.[60a] Depending on the eventual purpose and utilization of an architecture representation, it is decisive, for example, whether a view on the systems software components and related algorithms or a view on the electrical wiring of several hardware components is observed.[61a] In a nutshell, different perspectives resp. viewpoints facilitate different usage possibilities and a single architecture is not capable of covering all concerns of a system.[61a]

Bagschik et al.[61b] argue that holistic systems engineering not only requires the acquisition of diverse viewpoints on a system but also the description of correspondences between these views to allow traceability of system and sub-system properties.[61a] They discuss several viewpoint opportunities and extend established ones towards a capability based assessment of automated vehicles, particularly in the context of a safety analysis. It is stated that "in order to develop a safe system, it is indispensable to also define the dependencies between perspectives"[61c]. Based on this, they consider the skill and ability graph approach of their colleagues Reschka et al.[62] and Nolte et al.[63] and discuss how nodes in those graphs can be transfered from a behavioral to a technical view. According to them, a skill graph can thereby assist in elaborating functional to technical requirements by annotating enhancing information to certain nodes.[61d] The concept is transformed in a new so-called capability viewpoint, which is considered as an intermediate representation of a functional and component-focused viewpoint, e.g. a software or hardware viewpoint. Bagschik et al. finally claim that the tracing of a system's external visible behavior to properties of software or hardware components is enabled by such capability viewpoints. However, well-formulated relations between related viewpoints would be necessary to achieve this.[61d] The authors offer a selection of publications that they consider promising in this context, one of which is the approach by Dajsuren et al.[60b].

Dajsuren et al.[60b] initially discuss architecture views by elaborating existing architecture description techniques, particularly in the automotive sector, as well. Based on this, they further suggest a method to formally describe correspondence rules between architecture perspectives. The scope of their approach covers issues along structural views, especially between functional and software viewpoints. Besides, they address semantic inconsistencies between OEMs, which are regarded as those responsible for a functional view, and suppliers, which are often accountable for the software view.[60a] Several architecture relations are considered and an example for a "*refinement relation*" incl. correspondence rule is presented.[60c] Their overall aim is to improve consistency of architectural modeling, primarily on a semantic level. Factual benefits resulting from the use in approaches such as that of Bagschik et al.[61b] have yet to be demonstrated. Nevertheless, Bagschik et al.[61b] refer to Dajsuren et al.[60b] and abstractly include the suggestions in their capability viewpoint concept.[61d] Accordingly, with the help of such formalized correspondences, the traceability from software and hardware components towards

[60] Dajsuren, Y. et al.: Correspondence Rules for Architecture Views (2014), a: pp. 1-2; b: – ; c: p. 4.
[61] Bagschik, G. et al.: Architecture Framework for Automated Vehicles (2018), a: pp. 1-2; b: – ; c: p. 4; d: pp. 6-7.
[62] Reschka, A. et al.: Ability and Skill Graphs (2015).
[63] Nolte, M. et al.: Towards a Skill- and Ability-Based Development Process (2017).

the system's external behavior can be reached.[64a] Their understanding of an exemplary architecture framework, including a capability viewpoint and related correspondences is depicted in Figure 2-8.

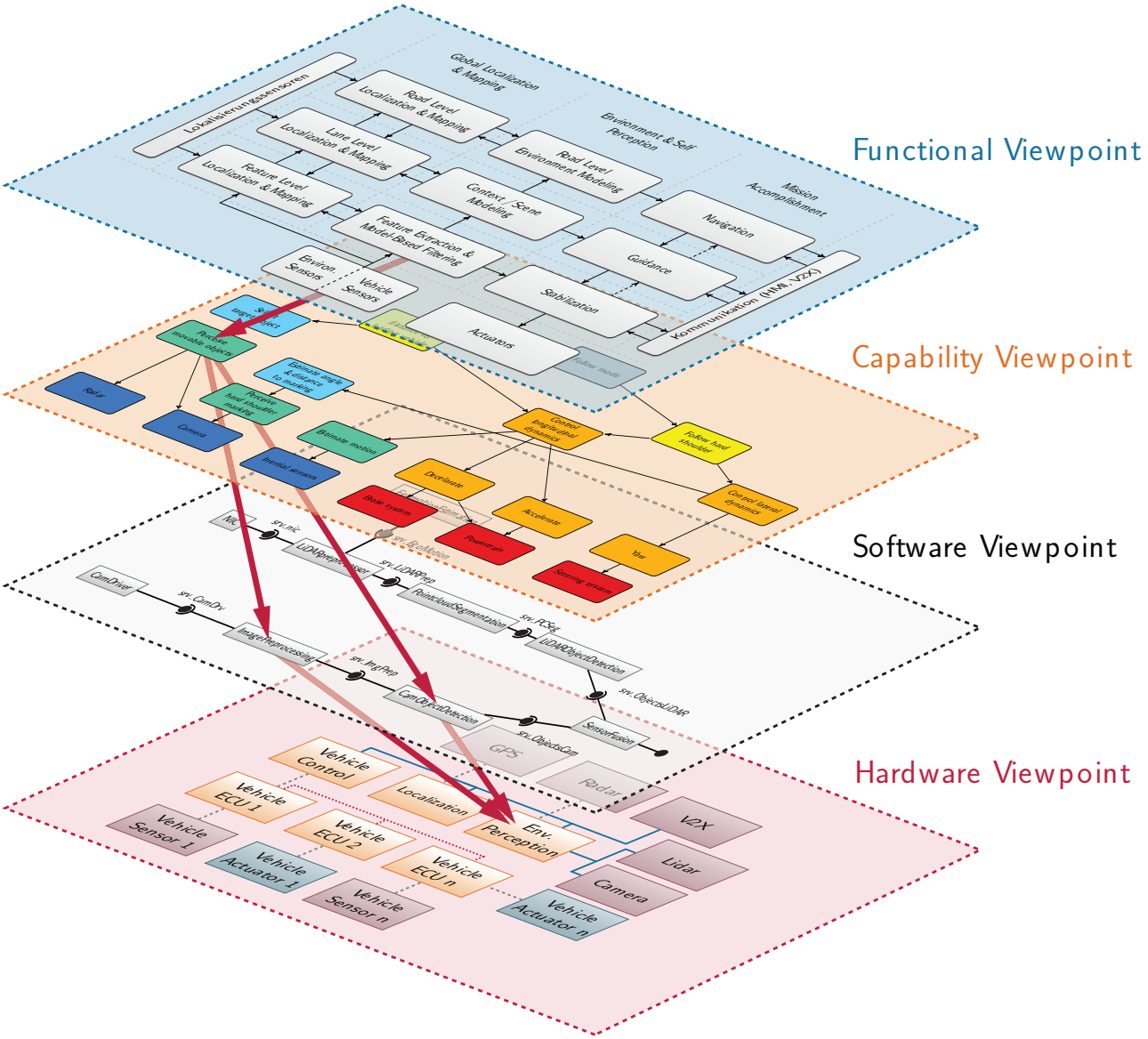


Figure 2-8: Architecture framework proposed by Bagschik et al.[61b]. The framework is composed of a functional, software, hardware and capability viewpoint. Dependencies resp. correspondences between viewpoints are indicated by red arrows.[64b]

[64] Bagschik, G. et al.: Architecture Framework for Automated Vehicles (2018), a: pp. 6-7; b: based on Fig. 6, p. 7.

2.5 Situation Awareness and Information Acquisition in Driving

In the following, a situation awareness model is reviewed to create a common understanding of information acquisition in driving for this work. How do drivers acquire information and how do they cope with driving situations? The study of this question is a fertile research area in the field of Human Factors and reveals parallels to functions inherited in systems for automated driving.[65a] Hereby, the term situation is not equivalent to the phrase in Section 2.3. Within the Human Factors discipline the term is used in a broader context.

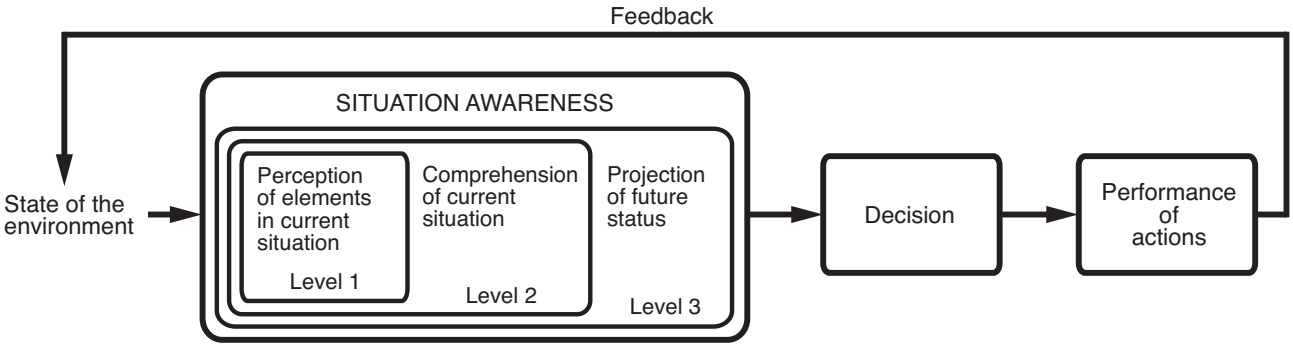


Figure 2-9: Simplified model of situation awareness in dynamic decision making suggested by Endsley[68a]. The overall task is conceptually separated into successive perception, comprehension, projection, decision and action steps.[68b]

Comprehensive receiving and processing of sensual information enables drivers to make decisions and maneuver appropriately according to various traffic scenarios. Many models concerning this process were suggested throughout history of Cognitive Science, Psychology and Human Factors. Moore[66] and Rumar[67], for example, have already highlighted the relevance and complexity of perception and attention processes at an early stage. Both authors view the driver as some kind of "traffic system information processor" [65b] and accentuate the importance of information search and selection processing in driving situations. Thereupon, Endsley[68a] proposes a model with greater reference to the variability of information being processed in such processes. She states that environmental as well as task conditions change constantly while driving and that drivers therefore have to carry out continuous decision making. This is referred to as situation awareness.[68c] A slightly simplified version of Endsley's model is depicted in Figure 2-9. Situation awareness is here differentiated into three levels. The first level is the perception of elements from the environment, e.g. road, traffic and vehicle conditions. The second level refers to the understanding of the current situation based on prior perceived information, thus, in this step the situation has to be given a meaning. Subsequently, in the third and highest level, foreseeing of the near future is reached by knowing and integrating the two previous

[65] Castro, C.: Human Factors in Driving (2008), a: p. 2; b: p. 9.
[66] Moore, R. L.: Human Factors Affecting Design of Vehicles and Roads (1969).
[67] Rumar, K.: The Human Factor of Road Safety (1982).
[68] Endsley, M. R.: Situation Awareness in Dynamic Systems (1995), a: – ; b: based on Fig. 1, p. 35; c: p. 34.

levels. In addition to these three key aspects of situation awareness, Endsley also points to further cognitive steps like decision making and the eventual performance resp. execution of actions.[69]

Comparable understandings of driving tasks spread in the area of advanced driver assistance systems and automated driving today. Not only in research but also within practical discourses, automotive engineers use similar terms for different aspects of driving tasks. An academic example, albeit a distant one, is the approach by Amersbach et al.[70a] concerning functional decomposing of driving tasks for test case generation. Their approach aims to reduce the approval effort regarding the safety verification of highly automated vehicles by a six-layer decomposition of the overall driving function. The derived layers are *Information Access*, *Information Reception*, *Information Processing*, *Situation Understanding*, *Behavioral Decision* and *Action*.[70b]

This work's approach makes use of the above suggestions to enhance the modeling and visualization of a system's architecture. The related concepts and explanations are to be found in Section 3.3.

[69] Endsley, M. R.: Situation Awareness in Dynamic Systems (1995), pp. 36-37.
[70] Amersbach, C.; Winner, H.: Functional Decomposition (2017), a: – ; b: pp. 3-4.

3 Concept of Approach

This chapter explains the holistic concept of this work's approach. Initially, a high-level view of the endeavor is described in form of a meta model and three key challenges are extracted. Thereupon, the considered ontology setup is addressed by not only examining the utilized development methodology more closely but also discussing the ontology's terminology and scope. It is also stated in which way the resulting ontology resp. corresponding knowledge bases are utilized. Afterwards, a special manner of depicting system architectures as skill graphs, the so-called task-chain-pattern skill graph representation, is suggested and explained. Lastly, the concept for deriving dependency chains is presented as the core of this work's approach. Overall, an abstract view of developed concepts is given without going into specific examples. The practical implementation is subject of Chapter 4.

3.1 Meta Model

The goal is to derive dependency chains connecting particular circumstances in ontology based knowledge representations of driving scenarios with certain system components. These chains are meant to link diverse substructures which do not inherently exhibit dependencies. Therefore, a holistic view of the difficulties and a concept of an overall structure is necessary to solve the imposed task.

At this point the terminology proposed by Steimle et al.[71a], previously mentioned in Section 2.3, is taken into account. Their definitions have proven to be beneficial in this context, as they enable the partial illustration of the intended structure interrelations. Additionally, the offered UML diagrams facilitate a consequent way of illustrating the evolved concept meta model. Steimle et al.[71a] address two key parts involved here. On the one hand, they outline terms regarding scenario representations, particularly concerning the generation of scenarios for scenario based testing.

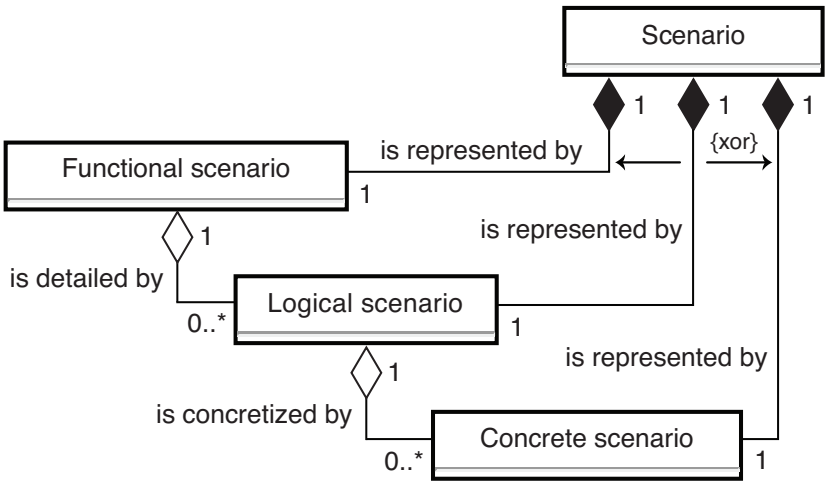


Figure 3-1: Terminology of scenarios with regard to different abstraction levels depicted as a UML diagram by Steimle et al.[71a] (translated from German). Scenarios are either represented as functional, logical or concrete scenarios.[71b]

[71] Steimle, M. et al.: Terminologie für den szenarienbasierten Testansatz (2018), a: – ; b: based on Fig. 2, p. 6.

On the other hand, definitions for the design and implementation phase which, among other things, include terms for system components, are suggested. Following, the consistency of their terminology and UML depictions is utilized to create a coherent meta model which complies with the state of the art understanding of related terms.

The first part is shown in Figure 3-1. As to be seen, the different abstraction levels of scenarios, as proposed by Menzel et al.[72], are presented. Accordingly, a scenario is represented as either a functional, logical or concrete scenario. In this work, the three levels of abstraction are included in the concept in order to be able to represent not only functional scenario circumstances but also various concrete parameter conditions.[73a] As a merger, the differently abstract scenarios are understood as a representation of the development vehicle's final field of application. Following, their merger is therefore referred to as the *operational scope* in the meta model. It represents one end of the intended dependency chains.

The second part is depicted in Figure 3-2. Here, Steimle et al.[73b] define terms related to their understanding of a specification. With reference to the ISO 26262, the specification is regarded as the starting point for the design and implementation phase. It is stated that the specification as a document specifies the requirements for the development object and its functional behavior at the vehicle level.

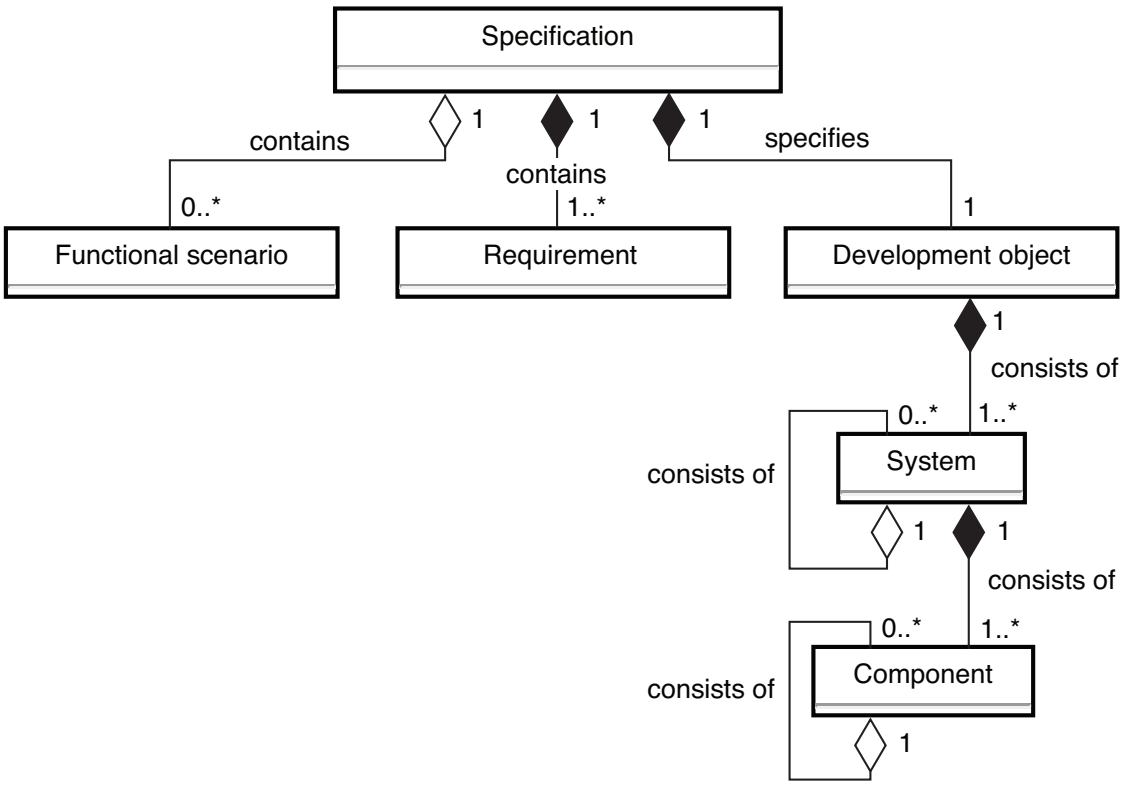


Figure 3-2: Terminology of the design and development specification context depicted as a UML diagram by Steimle et al.[73b] (translated from German). A specification contains several functional scenarios and at least one requirement. It specifies the development object of interest.[73c]

[72] Menzel, T. et al.: Scenarios for Automated Vehicles (2018).
[73] Steimle, M. et al.: Terminologie für den szenarienbasierten Testansatz (2018), a: pp. 8-9; b: –; c: based on Fig. 3, p. 8.

The latter is achieved by the aforementioned functional scenarios, which in turn are detailed and concretized through other abstraction levels in course of the development process. A requirement is understood as a required property or capability which a development object has to fulfill. The development object is defined as a system or group of systems. Furthermore, a system is meant to consist of other systems and several components, which again are composed of other components resp. dedicated hardware and software components.[74a] These terms are also incorporated into the meta model. The involved systems and components as a merger are further referred to as the system's *technical architecture*, which represents the other end of the intended dependency chains.

Following, the terminology by Steimle et al.[74b] is slightly extended to enable conclusions to be drawn from one end to the other. The merger of both prior explained UML diagrams including the already introduced umbrella terms *operational scope* and *technical architecture* are depicted in Figure 3-3. Additionally, a subdivision of the requirement term is suggested. For this, the three scenario abstraction levels are transferred into the requirement context and the sub-categories *functional requirement*, *logical requirement* and *concrete requirement* arise. As a result, the differently abstract scenario representations are understood to express the correspondingly abstract requirements.

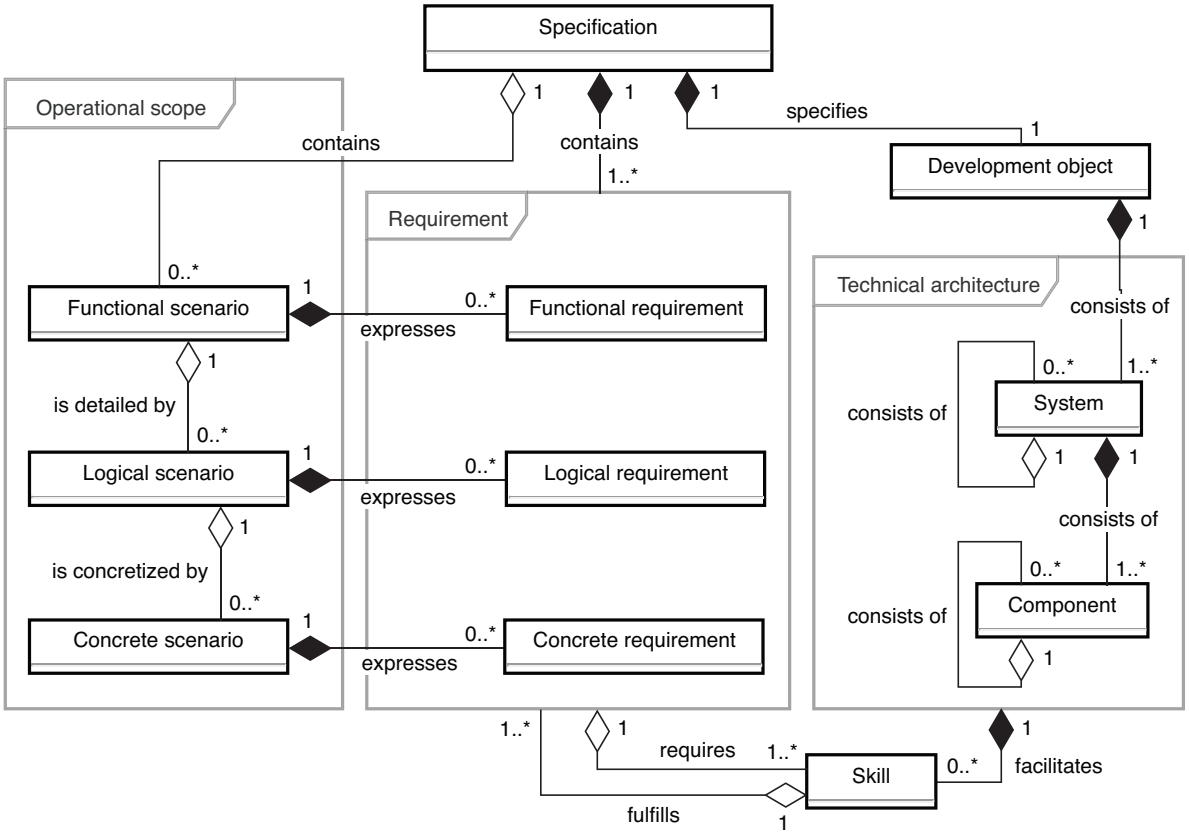


Figure 3-3: Merged terminology parts by Steimle et al. as a coherent UML diagram including additionally introduced umbrella and subdivision terms as well as a skill concept to enable the description of dependency chains between scenarios and system components.[74c]

[74] Steimle, M. et al.: Terminologie für den szenarienbasierten Testansatz (2018), a: pp. 8-9; b: –; c: based on Fig. 2 and Fig. 3, pp. 6-8.

Each scenario thereby expresses no, one or arbitrary many corresponding requirements, as stated by the cardinality information in the UML diagram. However, there is still no defined or self-evident relation between the requirements and the system's technical architecture. Thus, it is necessary to further extend the supplementary bypass to enable the description of dependency chains from one end to the other. For this purpose, the suggestions of Reschka[75] and Nolte et al.[76] (introduced in Section 2.4.1), particularly their concepts concerning a system's skills, are applied. Accordingly, skills describe activities of a technical system which have to be executed to fulfill defined goals of the system itself. Therefore, some parallels to the discussed terminology are recognizable. Not only are skills related to a technical system, including its technical architecture, but also the fulfillment of certain goals is mentioned. In this case however, the latter is interpreted as requirements. More precisely, in the meta model skills are understood to fulfill differently abstract requirements and to be facilitated by the technical architecture of a system. A requirement, in turn, makes certain demands resp. requires one or multiple particular skills. The originally formulated terms and the full supplementary introduced bypass is shown in Figure 3-3.

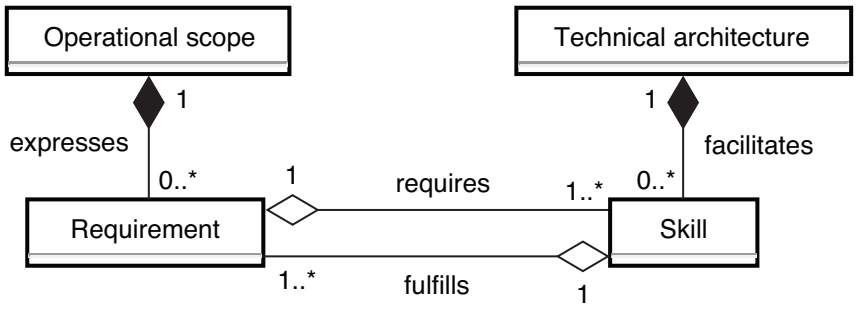


Figure 3-4: Meta model which illustrates the holistic concept of this work's approach. To achieve a derivation of the intended dependency chains, all depicted interrelations have to be represented appropriately.

The holistic meta model for the concept of this work's approach emerges by simplifying the former structure. It is depicted in Figure 3-4. After all, the concept leads to three key challenges, which are hereby clearly addressable:

1. Represent the operational scope in a way that enables the derivation of dependencies between scenarios and requirements.
2. Represent the system's architecture in a way that enables the derivation of dependencies between architecture sub-components and skills.
3. Accomplish an automatic derivation of dependencies between a representation of the operational scope and a representation of the system's architecture by utilizing 1. and 2. as well as providing the dependencies between requirements and skills.

The following sections of this chapter separately cover these three parts of the overall approach.

[75] Reschka, A.: Diss., Fertigkeiten- und Fähigkeitengraphen (2017).
[76] Nolte, M. et al.: Towards a Skill- and Ability-Based Development Process (2017).

3.2 Ontology Setup

As stated in the previous section, when implementing the derivation of dependency chains as proposed, the first key challenge is to represent the operational scope in a way, that enables the derivation of certain requirements. Additionally, the operational scope is meant to be composed of a set of scenarios. In this context, an ontology based representation offers promising advantages. Not only does this way of representing scenario knowledge offers versatile features like inferencing and querying capabilities (Section 2.2.1). It is also the subject of state of the art publications (Section 2.1). For these reasons an ontology for the representation of scenarios resp. the operational scope is developed. Following, the applied development methodology and decisions in regard to the ontology's scope as well as terminology are explained. For the implemented ontology, it is referred to Section 4.2.

3.2.1 Methodology

The development of an ontology is a complex endeavor with a great creative component.[77] Many decisions are to be made during the process and there is never a single correct way to achieve the intended purpose.[78a] Therefore, in order to support ontology developers and unexperienced researchers in particular, different development methodologies and guides have been proposed. General parallels and differences between such methodologies are outlined in Section 2.2.3. In this work, the methodology of Noy and McGuinness[78b] is applied, because they do not only take issues of several other methodologies into account but also specifically build on their practical experience. Besides, particularly beginners which are new in the field of ontology development are addressed by their guide.

Noy and McGuinness propose the consideration of three fundamental rules throughout the entire ontology development process. Following, each rule's consequences for this work's ontology development approach are discussed. The first rule is:

1. "*There is no one correct way to model a domain – there are always viable alternatives. The best solution almost always depends on the application that you have in mind and the extensions that you anticipate.*" [78c]

This statement underlines the importance of knowing exactly what the ontology is developed for. Accordingly, only if the ontology's eventual field of application is determined thoroughly, a high-quality representation of the domain's knowledge is made possible. To comply with the first rule, the intentions of this work's approach and the particular consequences for the developed ontology are specified in more detail. As a result, the following key intentions emerge:

- The ontology shall be able to represent driving scenarios mainly on a functional level but shall also enable to include logical or concrete scenario information.

[77] Mizoguchi, R.: Ontology Development, Tools and Languages (2004), p. 62.
[78] Noy, N. F.; McGuinness, D.: Ontology Development 101 (2001), a: p. 3; b: –; c: p. 4.

- A particular subset of scenarios, namely "Driving in the lane with dynamic objects" and "Crossing an intersection with traffic lights or traffic signs" shall be representable by the ontology.
- The ontology shall contain possibilities to model particular driving scenarios which shall finally be linked with some form of system components.
- Eventually, an automatic derivation of dependencies between the modeled circumstances and the system components shall be facilitated.

These intentions are kept in mind during the whole ontology development process and constantly affect design decisions. Especially the scope of the developed ontology is determined by the specified intentions, as explained in Section 3.2.2.

Noy and McGuinness further propose to initially develop a first version of an ontology before revising it over and over afterwards. Revisions shall thereby be enabled by testing and debugging the ontology in an application or through discussions with experts. Because of these consecutive development passes, they consider an ontology to have a lifecycle and formulate their second rule:

2. "*Ontology development is necessarily an iterative process.*" [79a]

For this work, the consequence of the second rule is an iterative organization of the practical implementation approach. To achieve this, initially, not only a basic ontology is developed but also the other required parts of the overall framework are laid out. Thereupon, everything is linked together on trial and the interaction within the resulting construct is observed. Then, based on emerging issues and further intended functionalities, the next iteration run is planned and every part, including the ontology, is corrected resp. improved. This process is repeated until all key demands on the eventual application are met.

The third rule is:

3. "*Concepts in the ontology should be close to objects (physical or logical) and relationships in your domain of interest. These are most likely to be nouns (objects) or verbs (relationships) in sentences that describe your domain.*" [79a]

Noy and McGuinness hereby explicitly point out, that the naming of concepts in the ontology is not arbitrary. The rule states that it is desirable to name classes and relations in the developed ontology according to real-life objects and relations between them. Additionally, the authors continuously offer particular instructions on how to define classes and explain how to choose the right terminology after all. Following this rule, the terms for the classes and relations of this work's ontology are chosen conscientiously with regard to real objects and circumstances in driving scenarios. The approach for the ontology's terminology is further outlined in Section 3.2.3.

[79] Noy, N. F.; McGuinness, D.: Ontology Development 101 (2001), a: p. 4.

3.2.2 Scope

The development of the ontology starts by defining its domain and scope. Both emerges by consideration of the previously specified intentions. It becomes apparent that the representation of a particular sub-set of driving scenarios is the domain of the ontology. Defining the scope, in turn, is more complicated and leads to further questions, because it is not trivial how much information has to be embedded and which level of detail has to be provided. Noy and McGuinness hereby suggest to sketch a list of so-called "*competency questions*" [80a]. Accordingly, these questions are to be answered by a knowledge base which is based on the developed ontology. Thus, competency questions not only verbalize but also limit the ontology's scope.

For this work, following competency questions are formulated:

- Does the system drive towards an intersection?
- What kind of intersection is it?
- Is there a traffic regulating element associated with the intersection?
- Which other traffic participants enter resp. leave the intersection?
- Is the system in a maneuver with another traffic participant?
- What is the distance between the system and another object?
- What is the relative speed between the system and another object?
- Are two objects on the same lane or road?
- Which differences appear between two consecutive scenes in a scenario?

Looking at this list of questions, it is noticeable that the ontology's scope is hereby characterized by particular examples which are based on the previously mentioned subset of scenarios. All questions are either considering the scenario "Driving in the lane with dynamic objects" or the scenario "Crossing an intersection with traffic lights or traffic signs". The chosen scenario subset therefore defines the ontology's scope thematically. However, the limit of the scope is defined by the questions' level of detail and the amount of questions. The more questions are asked and the more detailed they are, the wider the scope needs to be chosen. The formulated competency questions provide assistance in the overall development of the ontology.

3.2.3 Terminology

In regard to which terms and phrases to use in the developed ontology and how they are arranged to each other, the methodology by Noy and McGuinness suggests five successive steps.[80b]

The first step is to consider the reuse of existing ontologies. This aims not only to reduce development effort but also facilitates to comply with common standards and other terminologies if necessary. For

[80] Noy, N. F.; McGuinness, D.: Ontology Development 101 (2001), a: p. 5; b: pp. 5-9.

this work, in fact, already existing concept definitions are included in the ontology development. On the one hand, the terms and relations regarding driving contexts by Steimle et al.[81] (see Section 2.3) are considered to meet state-of-the-art standards in this context. On the other hand, a recent publication by the *National Highway Traffic Safety Administration* (NHTSA)[82] is included. The latter provides a very comprehensive collection of terms, however, serves hereby more as an orientation for terms that are not part of Steimle et al.'s terminology. In this way, the generation of an inconsistent composition of terms is prevented. Both sources are no ontologies in a literal sense, but support the ontology development process by offering established phrases which are then adopted.

The second step is to enumerate any important terms in the ontology. Here, any terms resp. concepts that a developer intends to deal with afterwards are understood as important. The goal is to "*get a comprehensive list of terms without worrying about overlap between concepts they represent, relations among the terms, or any properties that the concepts may have, or whether the concepts are classes or [relations]*" [83]. This process challenges developers to creatively reflect on the purpose of the intended application. In the context of this work, the resulting list is composed of terms from the first step as well as additional terms which are in some way related to the intended use of the ontology. Neither relations between terms nor any related properties are described explicitly. This list is meant to provide an initial basis for the subsequent steps.

The steps three and four are closely intertwined[83], which complicates to do one of the steps at a time. Nevertheless, in order to provide clarity, they are explained separately in the following.

The third step involves the definition of classes and the development of a class hierarchy. Different methods, depending on the developer's view of the domain, are suggested. Top-down, bottom-up and combined approaches are outlined by Noy and McGuinness.[83] Regardless of the approach, the previously created list of terms serves as a basis. A top-down approach is meant to start with the most general concepts in the list before continuing with specialized concepts afterwards. In contrast to that, a bottom-up approach aims to start with the most specific concepts in the list before grouping them under more general terms. In the combined approach, both methods are applied simultaneously. Here, the concepts presumably most relevant in regard to the intended use of the ontology are defined first without worrying about their generality or abstractness. Regarding this work's ontology development process neither a systematic top-down view of the domain is available nor is the relevance of the most specific concepts known. Because of this, the combined approach is chosen. Eventually, to define classes, only terms that describe independently existing objects in reality are selected. Subsequently, the emerged classes are hierarchically arranged as super-classes and sub-classes. The resulting taxonomy thereby indicates hierarchical relations between the selected class concepts, but it does not include information about further relationships between any classes or other properties.

The fourth step transforms the emerged taxonomy into an actual ontology by describing the internal structure of the classes, thus, particularly the relations between them. The remaining terms of the

[81] Steimle, M. et al.: Terminologie für den szenarienbasierten Testansatz (2018).
[82] Thorn, E. et al.: Automated Driving System Testable Cases and Scenarios (2018), pp. 31-40.
[83] Noy, N. F.; McGuinness, D.: Ontology Development 101 (2001), p. 6.

priorly generated list are likely to be properties or relations and are used hereby.[84] Each of these concepts has to be attached to suitable classes. Because sub-classes inherit all properties of their superior classes, properties shall always be attached to the most general classes.[84] Here, the intertwining of step three and four becomes apparent. When reusing the terminologies of other publications in the context of this work, as described in step one, it is noticeable that in most cases only the classes themselves are defined and not their properties or relationships with each other. The challenge is therefore to define the right relationships and characteristics of the adopted concepts without compromising the consistency of the terminology.

After completing the first four steps, a simple ontology results. In the fifth step, the developer then further enhances the ontology by adding more details to the defined relations and properties. Depending on the ontology language, the integration of differently complex details is enabled. The *Web Ontology Language* (OWL), which is used in this work and explained in Section 2.2.2, for example, facilitates the specification of information about cardinalities and value types as well as the description of property domains and ranges.

By repetition of these steps, especially steps three, four and five, the amount of specified concepts in the terminology grows and the resulting ontology is becoming more and more expressive. The result of this process in the context of this work is presented in Section 4.2.

3.2.4 Utilization

At this point a short description of the way in which the developed ontology is eventually utilized is given. The ontology resulting from the development process initially specifies nothing more than a mere conceptualization of domain knowledge. The class concepts including their properties and relations are defined by OWL and SWRL further enables the specification of universal axioms. Thus, only implicit knowledge about concepts and their interdependencies is captured.

In order to model the intended operational scope, however, it is necessary to explicitly describe functional, logical and concrete driving scenarios. For this purpose, the ontologically specified concepts are instantiated. This means that, on the basis of the knowledge modeled by the ontology, explicit instances of classes, so-called individuals, are formed. As already described in Section 2.2.1, knowledge bases are the result of this process. In the context of this work, a set of such knowledge bases is ultimately used to represent the operational scope. The knowledge about explicit instances therein is extracted to identify specific circumstances of particular interest in the operational scope. This is further described in Section 3.4.

[84] Noy, N. F.; McGuinness, D.: Ontology Development 101 (2001), p. 8.

3.3 Task-Chain-Pattern Skill Graphs

The second key challenge of this work's approach, as outlined in the beginning of this chapter, is to represent a system's technical architecture in a way that enables the derivation of dependencies between architecture sub-systems resp. components and the system's skills. In order to address this challenge, the suggestions of Reschka[85] and Nolte et al.[86a] (introduced in Section 2.4.1) for representing skill and ability graphs are utilized and extended. Following, it is described which parts of their concept are adopted and which further demands on an architecture representation emerge in the context of this work. As a result, it is suggested to represent architectures by so-called *task-chain-pattern skill graphs* and the modeling guidelines for this type of graphs are outlined.

3.3.1 Provenance and Demands

As previously mentioned, the following introduced task-chain-pattern skill graphs are based on the suggestions of Reschka[85] and Nolte et al.[86a]. Reschka and Nolte et al. aim to provide a basis for the development phase as well as the operation of self-aware automated vehicles. Therefore they propose two coherent concepts, skill graphs and ability graphs, wherein each intends to address a particular application (see Section 2.4.1). In summary, skill graphs provide insights into how skills interrelate with each other, sensors or actors and are able to visualize not only functional but physical redundancies. Ability graphs, on the other hand, include abilities, which are instantiated skills, and represent implemented technical solutions during operation. They comprise certain performance levels and are meant to be used for online monitoring.

For this work, the skill graph approach is considered, because it is designated to support the concept phase of the development process and because it enables the representation of chains between superior functional system activities and technical components like actors and sensors. Additionally, the concept of performance metrics is applied. Not to express performance levels of a vehicle during operation, but to be able to formulate concrete requirements for system components. Following, the adaption and extension of the skill graph concept is explained. In which way the idea of performance metrics is included in this work's approach is described in Section 3.4.2.

Nolte et al. additionally suggest to classify the skills in a skill graph according to their task type. The four categories main, sense, plan and act are offered and each skill is colorized correspondingly.[86b] During the use of skill graphs in the context of this work, however, a closer association with successive task steps is desired, because it seems to ease the creation of such graphs when a system's capability is described as a sequence of consecutive skills. As stated by Nolte et al., the basic idea of skill graphs is to "*model the system along the human driving task*" [86c], but they merely include this in their representation. Although the edges of the graphs provide information about which skills determine each other, the temporal relationship between skills resp. their chronological order is not always obvious.

[85] Reschka, A.: Diss., Fertigkeiten- und Fähigkeitengraphen (2017).
[86] Nolte, M. et al.: Towards a Skill- and Ability-Based Development Process (2017), a: – ; b: p. 4; c: p. 3.

Task-chain-pattern skill graphs intend to address this issue by an alternative arrangement of the skills. The basic concept is depicted in accordance with Reschka et al.[87a] and shown in Figure 3-5. At this point, the situation awareness model in dynamic decision making by Endsley[88], introduced in Section 2.5, is taken into account. According to Endsley, a driving task is dividable into three situation awareness steps and two additional subsequent steps. Based on her suggested steps, the skill categories *Perception*, *Comprehension*, *Prediction*, *Decision* and *Action* are proposed for task-chain-pattern skill graphs. On this basis, all skills in a graph are assigned to one of these five categories.

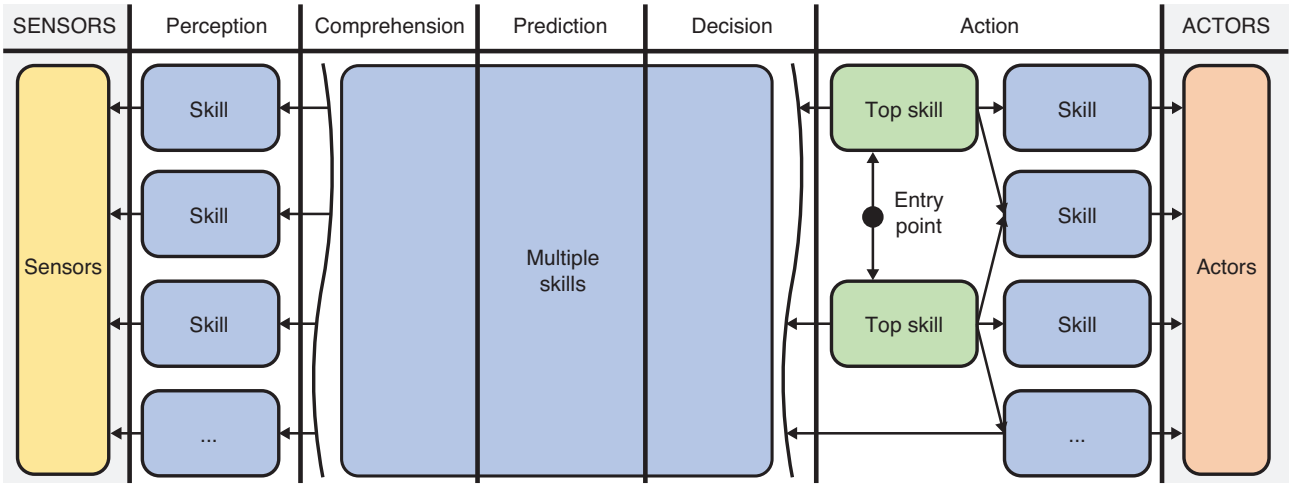


Figure 3-5: Basic task-chain-pattern skill graph representation in accordance with the skill and ability graph concepts of Reschka et al.[87a]. Skills are arranged in regard to five successive steps of driving tasks. Sensors are represented on the left and actors on the right.[87b]

In addition to the adopted concept of skill nodes in general, other key features of the original skill graphs are inherited in the new concept. First of all, only directed acyclic graphs result by applying both modeling techniques. Secondly, edges of the graph still represent the dependency between superordinate and subordinate skill nodes.

Task-chain-pattern skill graphs fulfill the previously stated demands on the representation of the system's technical architecture. In order to facilitate the adaption of the presented approach, developed modeling guidelines are explained in the following.

3.3.2 Modeling Guidelines

A set of modeling guidelines is developed to guarantee a consistent creation of task-chain-pattern skill graph models. Following, the immanent node concepts are therefore explained in more detail and generic modeling rules are outlined.

A task-chain-pattern skill graph always models a so-called *main skill*, consists of seven columns and contains five different node concepts. The regular *skill nodes* represent skills according to the

[87] Reschka, A. et al.: Ability and Skill Graphs (2015), a: –; b: in accordance with Fig. 2, p. 4.
[88] Endsley, M. R.: Situation Awareness in Dynamic Systems (1995).

definition of Nolte et al.[89] and are located in the five middle columns. *Sensor nodes* are located in the first column on the far left and represent the concrete sensor components of the system's technical architecture. Similarly, *actor nodes* represent concrete actor components of the system and are located opposite to the sensor nodes in the last column on the far right. The *entry point node* is comparable to an entry point in state or flow charts and marks the beginning of the graph. It hereby represents the one superordinate skill of interest resp. the skill which the graph deals with. Because this skill always constitutes an executable action, the entry point node is arranged in the action column. *Top skill nodes* represent observable behaviors of the system and are located at the left side of the action column. This means that no other action nodes depend on them. They therefore represent superior action skill nodes.

A task-chain-pattern skill graph may be interpreted in two different ways. Either the connected skill nodes are viewed from left to right or the directed edges are followed from the entry point node to the ends of the graph. The first way represents the chronological order of temporally successive skills and depicts chains from the sensing of environment information by the system's sensors to eventual actions of actor components. The second way represents hierarchical sequences, hence, illustrates the dependencies between skills from the superior main skill down to all sensors and actors.

As previously explained, regular skill nodes are classified into one of five tasks steps resp. columns. Their dependencies are expressed by connecting the nodes with directed edges. Edges that protrude from one column to the other always lead in a given direction. Between the first six columns, the edges are directed from right to left. From the nodes in the action column over to the last column edges always lead from left to right. It is therefore not possible to model edges in the direction of the action column. Within a column, the horizontal displacement of nodes expresses the extent to which the skill nodes follow each other chronologically. Any number of skill nodes with different levels of detail is modeled. Therefore, differently detailed or simplified representations of one and the same main skill are realizable. Sensor and actor nodes form the ends of the directed graph, hence, they only have incoming edges. The top nodes, on the other hand, only have outgoing edges, apart from the connections to the entry point node. Edges might be modeled by spreading over several columns without passing through any skill nodes. If the same connection is already represented by a path with skill nodes and edges, the edge that protrudes over several columns is considered superfluous and therefore removed. However, if the edge represents the only path from one skill node to another, it is retained. In this case, a simplified modeling variant of the main skill is assumed.

These modeling guidelines are to be applied when developing task-chain-pattern skill graph representations of a system's architecture. An explicit example is presented in Section 4.3.

[89] Nolte, M. et al.: Towards a Skill- and Ability-Based Development Process (2017), p. 3.

3.4 Chain Derivation Engine

The third key challenge regarding this work's concept is to accomplish an automatic derivation of dependencies between a representation of the operational scope and a representation of the technical architecture by utilizing the previously outlined concept parts. The hereby intended results are dependency chains reaching from particular circumstances in ontology based scenario representations across system skills to components of the system's architecture. In order to achieve the formulated intention, a so-called *chain derivation engine* is developed. The chain derivation engine embodies the core of the presented approach and is composed of semantic rules combined with specific arithmetic expressions. Its mechanisms are explained in the following. First, the structure of the included semantic rules and their utilization for querying ontological knowledge from scenario representations are explained. Secondly, the embedding of performance parameters with the help of specific arithmetic expressions is described. Again, at this point, particularly the developed concept is described. An exemplary implementation of the chain derivation engine, in turn, is presented in Section 4.4.

3.4.1 Semantic Rules and Querying

Here, the modeled ontological knowledge bases, which represent the operational scope, are utilized. The knowledge bases constitute domain specific individuals and their semantic relationships in a formal and especially machine-readable manner. The mechanisms of the chain derivation engine take advantage of this. In order to extract the formally expressed knowledge, the RDF query language SPARQL[90], introduced in Section 2.2.2, is used. SPARQL enables the examination of any individuals and relationships in knowledge bases that correspond to the concepts of the developed ontology. For this purpose, the modeled knowledge has to be charged with particular queries. Eventually, anything that can be modeled by the developed ontology can then be queried by SPARQL.

The semantic rules of the chain derivation engine are based on these query capabilities of SPARQL. By adding expert knowledge, queries are individually tailored to dependencies between specific circumstances in scenarios and a corresponding skill, and formulated as rules. If a rule is fulfilled in relation to a certain operational scope, connections to the corresponding skill are indicated. Only circumstances of particular interest are addressed in order to illustrate the resulting dependencies. Dependencies modeled accordingly express functional requirements of the operational scope to the skills of the system. This is to be understood as:

> If a rule X is fulfilled, i.e. if the circumstances described thereby are present in a scenario of the operational scope, a skill Y needs to be facilitated by the system.

However, it is only possible to model functional dependencies in this way. In the following it is explained how the semantic rules are extended in order to be able to represent and derive more concrete dependencies, especially with regard to special system components and their parameters.

[90] Prud'hommeaux, E.; Seaborne, A.: SPARQL Query Language for RDF (2008).

3.4.2 Arithmetic of Performance Parameters

In order to be able to model more than functional dependencies, the semantic rules presented above are extended. The goal is to map the relations between explicit entities resp. individuals in scenarios and concrete components of the system's architecture.

For this purpose, special performance parameters are assigned to particular nodes in the task-chain-pattern skill graph representation of the architecture. For example, a sensor node receives the performance parameter "field of view" including a corresponding value or value range. The nodes are thus enriched with information about their intrinsic capabilities or technical limitations. In addition, the individuals and their relations in the developed knowledge bases are also modeled with corresponding, concrete values or value ranges. According to the different abstraction levels of scenarios defined by Menzel et al.[91], as outlined in Section 2.3, the functional scenarios are therefore transformed into concrete and logical scenarios. The representations of the scenarios might then include, for example, explicit positions, distances or velocities of individuals and relations as fixed values or value ranges.

In order to be able to process the additionally added information, the semantic rules of the chain derivation engine are extended by special arithmetic functions. These arithmetic functions aim to model any physical dependency in this context. Again, expert knowledge is required, since the intended correlations must be explicitly tailored to the conditions of the semantic rule. For example, the extent to which the aforementioned performance parameter "field of view" of a sensor is physically related to concrete distances or angles between certain individuals in the knowledge bases is modeled. But how does the chain derivation engine determine exactly which individuals are involved? Regarding this, the full capabilities of SPARQL are exploited. SPARQL not only enables to discover certain occurrences in ontology based scenario representations. Similar to queries in the context of databases, SPARQL also offers the possibility to explicitly return the entities addressed by a certain query. With appropriate modeling of the rules, exactly those individuals in the knowledge base are returned which are relevant for solving the arithmetic function.

By extending the rules, not only functional, but also the desired logical and concrete requirements of the operational scope on the skills of the system are expressible. Finally, dependency chains are enriched by additional conditions, namely the fulfillment of the arithmetic functions included in the rules. Thereby, it is possible to represent dependency chains reaching from particular circumstances in ontology based (functional, logical or concrete) scenario representations across system skills to components of the system's architecture and their performance parameters. An exemplary development and implementation of such semantic rules, including performance parameters and corresponding arithmetic functions, is outlined in Section 4.4.

[91] Menzel, T. et al.: Scenarios for Automated Vehicles (2018).

4 Implementation

In this chapter the practical implementation of the presented approach is pointed out. The aim of the implementation is to provide a proof of concept. As explained in the previous chapter, the concept of the approach consists of three different parts. To represent the operational scope, an ontology is developed, the presented task-chain-pattern skill graphs are used to represent the system's architecture, and eventually everything is set in relation to each other by the chain derivation engine. Initially, it is described how these partial components are integrated into a software framework in the course of the implementation. Thereupon, the individual parts and their explicit realization are addressed in more detail. First, the implementation of the operational scope, represented by the developed ontology and relating knowledge bases, is presented. Subsequently, it is explained how the system's architecture resp. the task-chain-pattern skill graphs are implemented. Then, the implementation of the chain derivation engine is outlined. The different parts are illustrated by explicit and coherent examples.

4.1 Software Framework Structure

The three parts of the concept discussed in Chapter 3 are developed and implemented in different ways, but in accordance with each other. In the following, the implemented union of the individual digital artifacts is referred to as the *software framework*. A simplified UML class diagram of this framework is shown in Figure 4-9. The complete version of the class diagram includes further classes, attributes as well as methods and is to be found in Appendix A.1. This diagram is not comparable to the UML diagrams presented in previous chapters. It does not represent terminological statements, instead it symbolizes the internal structure of the implemented software framework.

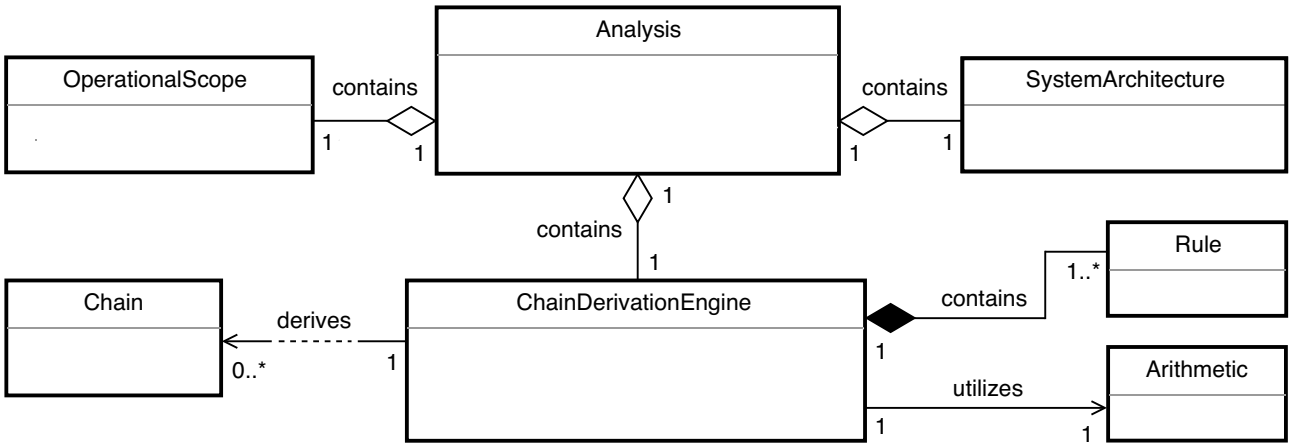


Figure 4-1: Simplified UML class diagram of the implemented software framework. The classes OperationalScope, SystemArchitecture and ChainDerivationEngine represent the three conceptual parts and are part of the Analysis class. The ChainDerivationEngine class contains instances of the Rule class, utilizes an respective Arithmetic and enables the derivation of dependency Chains.

As depicted in Figure 4-9, the operational scope, the system's architecture and the chain derivation engine are modeled as individual classes. The three digital artifacts are implemented separately and added to the overall framework. An Analysis class is introduced to unify the individual parts and realizes the intended analysis functionalities. Both the OperationalScope and the SystemArchitecture serve as information sources for instances of the Analysis class. The ChainDerivationEngine class is considered as the algorithmic core of the Analysis class and contains a set of semantic Rules and a set of arithmetic expressions collected in an Arithmetic object. Eventually, an instance of the ChainDerivationEngine class enables to derive the intended dependency Chains.

Following, the implementation of the three key parts, being OperationalScope, SystemArchitecture and ChainDerivationEngine, are explained in detail. Thereby, particular, interrelated examples are offered to give a profound insight. The overall software framework is implemented in Python. Each of the subsequent sections therefore also refers to certain packages or interfaces, which enable the integration of the individual digital artifacts.

4.2 Operational Scope

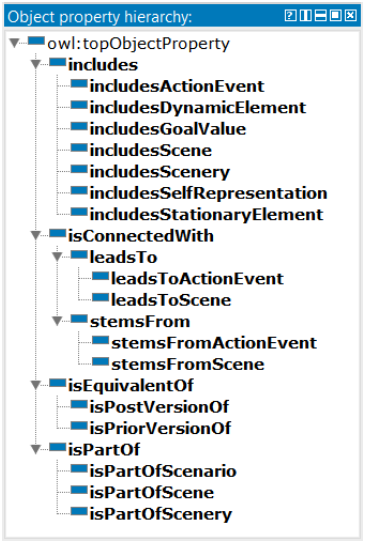
The concept of the operational scope aims at representing driving scenarios based on a developed ontology. Section 3.2.1 describes the methodology used to develop this ontology and how both its terminology and scope are elaborated. The concepts resulting thereby are then formalized and stored in a machine-readable way in order to enable their continuing utilization in the software framework. For this purpose, the open source ontology development tool and editor *Protégé*[92] is used, because it does not only provide all required functionalities but is also recommended in the context of the applied methodology by Noy and McGuinness[93]. For an overview of the features and an explanation of how the editor works, it is referred to the *Protégé* wiki[94].

During the implementation, all intentions outlined in Section 3.2.1 are pursued continuously. The terminology of Steimle et al.[95] therefore serves as a starting point for the specification of the terms in regard to general driving scenarios. In *Protégé* the class concepts based on this are first specified as a hierarchy resp. taxonomy. Additionally, the properties and relations of the concepts are modeled. The resulting ontology classes and relations are depicted in Figure 4-2 as excerpts from *Protégé*.

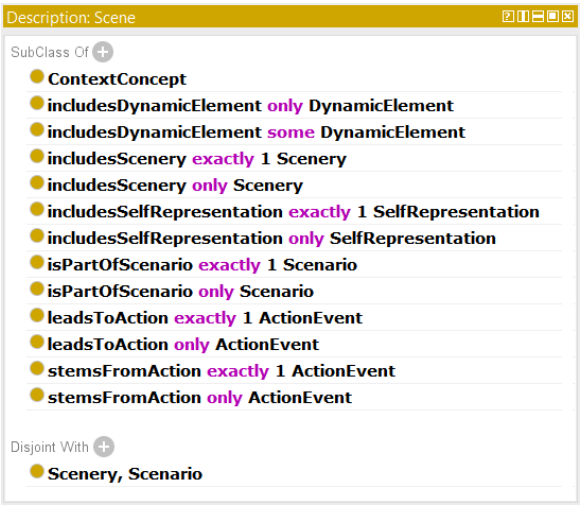
Figure 4-2 (a) shows how the terms proposed by Steimle et al.[95] are arranged hierarchically. In *Protégé*, all specified classes are subordinated to the owl:thing class. The terms Scenario, Scene and Scenery are regarded as Context Concepts. DynamicElements and StaticElements are integrated into a common ContextElement class. The GoalValue class comprises the two subclasses PermanentGoalValue and TransientGoalValue. The modeled superclass-subclass relationships are to be understood as "is a kind of"- relations.

[92] Stanford Center for Biomedical Informatics Research: Protégé (2016).
[93] Noy, N. F.; McGuinness, D.: Ontology Development 101 (2001).
[94] Stanford Center for Biomedical Informatics Research: Protégé Wiki (2016).
[95] Steimle, M. et al.: Terminologie für den szenarienbasierten Testansatz (2018).

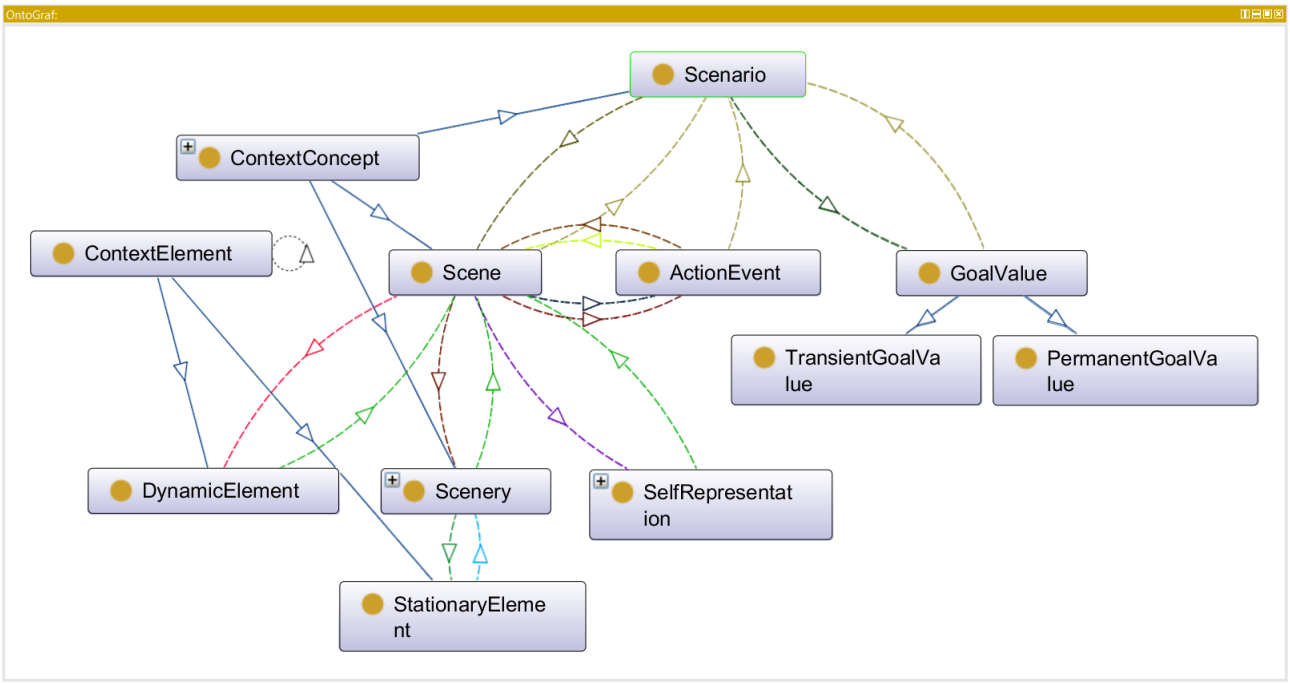
(a) Class hierarchy



(b) Relation hierarchy



(c) Scene class description



(d) *OntoGraf* visualization

Figure 4-2: Specified ontology classes and relations based on the terminology of Steimle et al.[96] as excerpts from the *Protégé* editor. Figure (a) shows the hierarchical class structure and Figure (b) depicts the specified relations. The more detailed modeling of a class is presented in Figure (c) using the example of the Scene class. Figure (d) shows a graphical representation of the ontology visualized by *OntoGraf* [97]. Here, the relations between classes are illustrated as arrows. The solid lines indicate "is a kind of"- relations and the dashed lines represent the various specified relations between classes. In order to maintain comprehensibility, labels of relations are not displayed.

[96] Steimle, M. et al.: Terminologie für den szenarienbasierten Testansatz (2018).
[97] Falconer, S.: OntoGraf (2013).

In Figure 4-2 (b) the specified relations are displayed. The includesScene relation, for example, enables to model scenarios which contain scenes. Accordingly, all relation concepts formulated by the terminology are implemented. In order to increase the expressiveness of the ontology, for some terms the inverse phrase is specified, too. For example, the isPartOf relation represents the inverse equivalent to the includes relation. The leadsTo and stemsFrom relations are used to model concatenations of several scenes and actions or events. The isPriorVersionOf and isPostVersionOf relations are also related to this. They express which context elements of a scenario represent time-shifted equivalents of each other in consecutive scenes. The specification of the relations that apply to particular classes, i.e. includesScene, also increases the expressiveness of the eventual ontology.

The way of how classes are enriched with information by the defined relation concepts is shown in Figure 4-2 (c) using the Scene class as an example. In *Protégé* this is realized by so-called "SubClass Of"- descriptions. Here, it is not only specified that a Scene is always a ContextConcept, but for example also that it must only contain exactly one Scene. The repetitive notation with two formulations for each relation expression explicitly specifies both the desired cardinality property and the class of the related concept. The figure also shows how information about disjunctive classes is stored. In the example it is formulated that a Scene is never a Scenery or Scenario.

Figure 4-2 (d) additionally presents a graphical representation of the specified classes and relations. The visualization is realized by the *OntoGraf* [97] plugin, a support tool for interactively navigating the content of ontologies, embedded in *Protégé*. When compared with Figure 2-6, it becomes clear that the depicted ontology models the exact concepts of the underlying terminology. The inverse relation constructs between certain classes become apparent, too. Hereby, the labels of the relations are not listed in the graph in order to maintain comprehensibility.

However, the terminology by Steimle et al.[98] only serves as a basic structure in the modeling of this work's ontology. As elaborated in Section 3.2.1, the ontology's scope must include further classes and relationships to enable the formulation of all intended interrelationships. The goal, however, is not to enable the modeling of all traffic scenarios imaginable, but merely to address the previously explained subset consisting of two exemplary scenarios. Even this does not entitle to a full description of all classes and relations imaginable in the context of these two scenarios.

The following scenario subset is addressed:

Scenario 1: Driving in the lane with dynamic objects.
Scenario 2: Crossing an intersection with traffic lights or traffic signs.

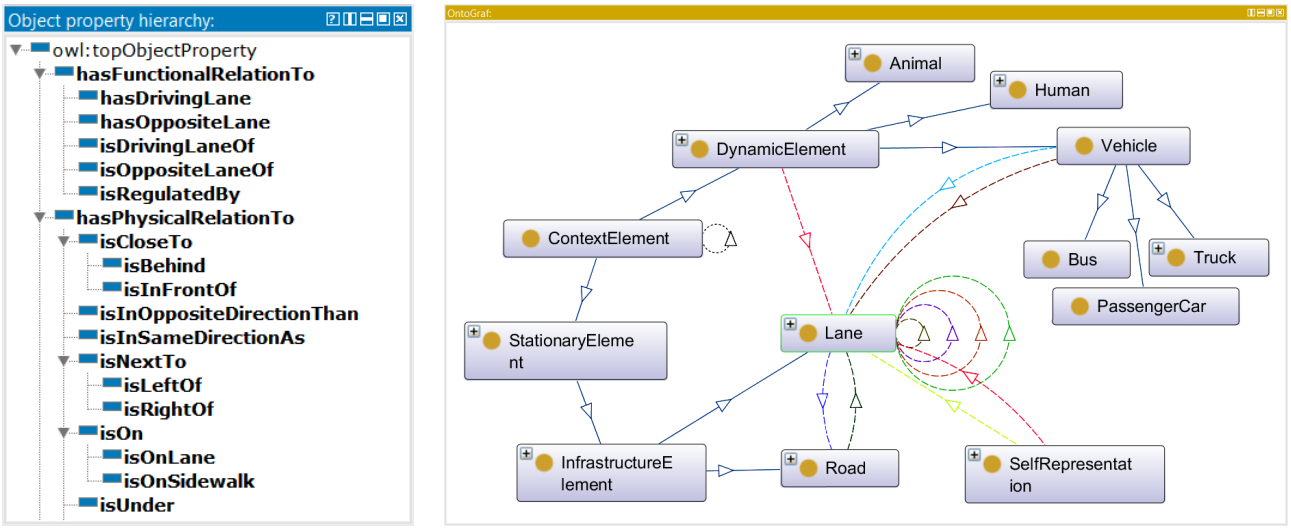
The textual description of the two scenarios already reveals information about which additional classes have to be added to the ontology. The first scenario mentions the term lane and refers to dynamic objects, whereas the second scenario at least requires classes that model the concepts of intersections, traffic lights and traffic signs. Therefore, these class concepts have to be specified in the ontology. Additionally, further associated concepts, i.e. relations between the additional classes, are required by both scenarios.

[98] Steimle, M. et al.: Terminologie für den szenarienbasierten Testansatz (2018).

In the following, the final ontology developed in the context of this work is discussed. Since the representation and explanation of all concepts of the entire ontology is too extensive, individual sections of the ontology are examined. The discussion of these sections aims at presenting the class and relation concepts required by the scenario subset. For the sake of completeness, the entire ontology is represented by comprehensive excerpts from *Protégé* in the Figures A-2 and A-3 in Appendix A.2.

The first scenario deals with driving in the lane as well as the interaction with foreign, dynamic objects. Figure 4-3 shows the specified concepts. While Figure 4-3 (a) shows a hierarchical list of the relations, Figure 4-3 (b) illustrates relations of additional classes to each other. Both representations are limited to a selected section of the entire ontology.

As a subclass of the StationaryElement class the InfrastructureElement class is introduced. It contains the classes Road and Lane, which are related. A Road always contains at least one Lane and instances of the Lane class are always part of a Road instance. Furthermore, the Lane class has relations between two instances of the Lane class itself. Thus it is possible to specify that Lanes are in physical relation, i.e. next to each other, as well as that they run in the same direction or even in the opposite direction. The latter is considered a functional relation. Additionally, the already specified DynamicElement class is extended by the isOnLane relation as well as additional subclasses. The figure shows examples of the Animal, Human and Vehicle classes including further subclasses. Between the Vehicle and Lane class the relations hasDrivingLane and hasOppositeLane are specified. These describe whether a Lane is heading in a Vehicle's direction of travel or in the opposite direction. The same relation is also specified between the SelfRepresentation and Lane class.



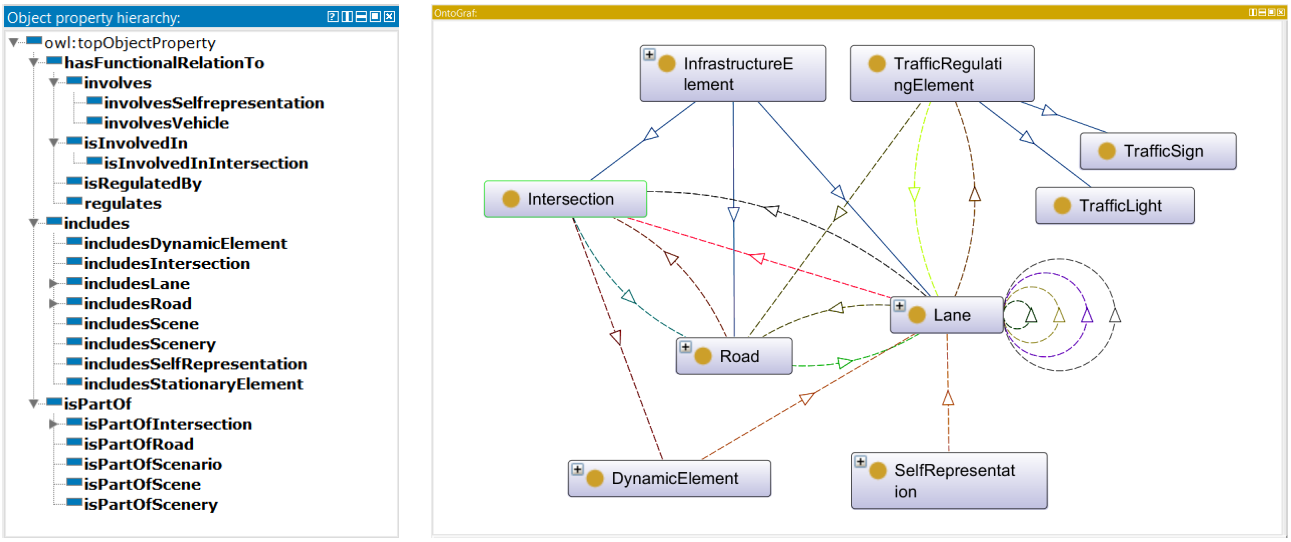
(a) Relation hierarchy

(b) *OntoGraf* visualization

Figure 4-3: Specified ontology classes and relations in regard to scenario 1 (Driving in the lane with dynamic objects) as excerpts from *Protégé*. Figure (a) shows the hierarchically structured relations. Figure (b) presents the ontology visualized by *OntoGraf* [99]. Solid lines indicate "is a kind of"- relations and dashed lines represent the specified relations. In order to maintain comprehensibility, labels of relations are not displayed.

[99] Falconer, S.: OntoGraf (2013).

In the context of the second scenario, reference is made to intersections and traffic-regulating lights and signs. The expressiveness of the ontology must therefore be further increased. For this purpose, additional elements of the infrastructure, traffic lights, traffic signs and corresponding relations are modeled. An extract of the resulting ontology classes and relations is shown in Figure 4-4. Again, the hierarchical listing of the relations is shown in Figure 4-4 (a) while the relations of the classes to each other are illustrated in Figure 4-4 (b).



(a) Relation hierarchy

(b) *OntoGraf* visualization

Figure 4-4: Specified ontology classes and relations in regard to scenario 2 (Crossing an intersection with traffic lights or traffic signs) as excerpts from *Protégé*. Figure (a) shows the hierarchically structured relations. Figure (b) shows a graphical representation of the ontology visualized by *OntoGraf*.

The classes Intersection, TrafficSign and TrafficLight are introduced and put into relation. A Intersection always consists of Roads, which in turn are composed of Lanes. These correlations are represented by the includesRoad and includesLane relations or corresdondingly isPartOfIntersection and isPartOfRoad relations. The classes TrafficSign and TrafficLight belong to the TrafficRegulatingElement class. This class is connected to the Lane class by the regulates resp. isRegulatedBy relation. DynamicElements, such as vehicles, are assigned to Intersections via the isInvolvedInIntersection relation. Through these specified concepts it is possible to model simple intersection scenarios with traffic regulating elements.

The actual modeling of driving scenarios based on the developed ontology is realized by instantiation. Thereby, the class and relation concepts are mapped by concrete instances. This results in knowledge bases which describe specific scenarios. In *Protégé* this is achieved by modeling *individuals*. As an example, Figure 4-5 shows a simple intersection scenario extracted from the *OntoGraf* plugin. The scenario modeled in this case is instantiated by the individual SCO_THIS. A scene SCN_0, a scenery SCY_0 and a self-representation SR_THIS are modeled as well. Moreover, with the individual I_0, there is an instance of the Intersection class. Four roads belong to the intersection, modeled by the

individuals R_1, R_2, R_3 and R_4. Each road then again consists of two instances of the Lane class. A special element is the individual TL_1. It instantiates the TrafficLight class and is in relation to the self-representation, a road and a lane. The modeled relation in this case express that the traffic light is approached by SR_THIS, that it regulates the lane L_1 and is located next to the road R_1. In this manner it is thus possible to model traffic scenarios through knowledge bases which are based on expert knowledge as well as the developed ontology.

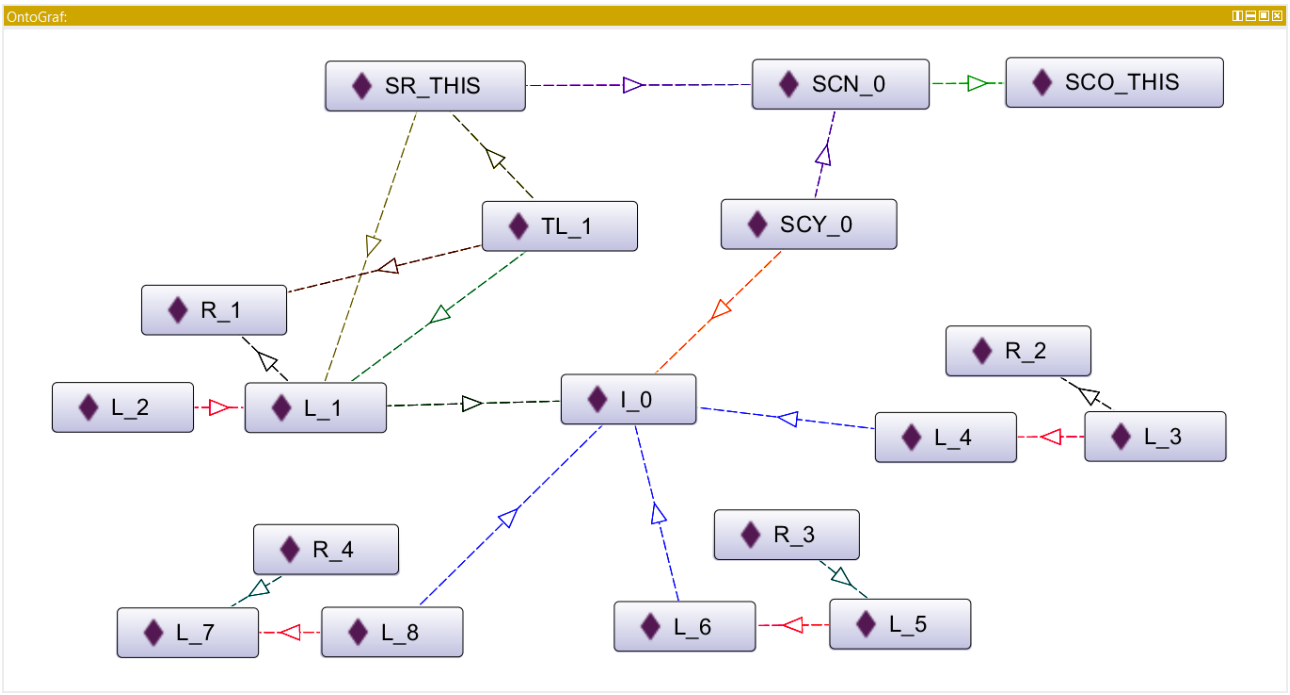


Figure 4-5: Exemplary intersection scenario knowledge base. Instances of different classes are specified and connected to each other. An intersection with four roads and a total of eight lanes is modeled. These individuals are part of a scenery, scene and scenario. Additionally, a traffic light and self-representation are represented.

The example from Figure 4-5, however, also illustrates something else. As to be seen, only a few relations are defined between the different individuals. Lane L_2, for example, does not appear to be directly connected to the intersection I_0. The intention in this knowledge base, however, is that each of the modeled lanes either leads into or out of the intersection. In this context, a definition is needed which expresses that if L_2 has a certain relation to L_1 and this relation is connected to I_0, then L_2 is also related to I_0. At this point the Semantic Web Rule Language explained in section 2.2.2 is utilized. SWRL facilitates to specify generally valid rules, so-called axioms, which are then added to the ontology to map more complex relations.

With respect to the aforementioned example, the following two SWRL rules are defined:

```
Lane(?L1) ^ Lane(?L2) ^ Road(?R) ^ Intersection(?I) ^
isPartOf(?L1,?R) ^ isPartOf(?L2,?R) ^ isInSameDirectionAs(?L1,?L2) ^
isIncomingLaneOfIntersection(?L1,?I)
-> isIncomingLaneOfIntersection(?L2,?I)
```

```
Lane(?L1) ^ Lane(?L2) ^ Road(?R) ^ Intersection(?I) ^
isPartOf(?L1,?R) ^ isPartOf(?L2,?R) ^
isInOppositeDirectionThan(?L1,?L2) ^
isIncomingLaneOfIntersection(?L1,?I)
-> isOutgoingLaneOfIntersection(?L2,?I)
```

Decoding the syntax reveals which relationships are explicitly modeled by the two rules. The first rule expresses that if two lanes are part of the same road, head in the same direction and one of these two lanes leads into an intersection, then the other lane must also lead into this intersection. The second rule is very similar yet quite the opposite. It states that if two lanes are part of the same road but head in different directions and one of these two lanes leads into an intersection, then the other lane must lead out of this intersection.

In this way, further axioms are specified and added to the ontology in order to model the correlations required from the scope of the ontology as accurately as possible. For this, *Protégé* offers another

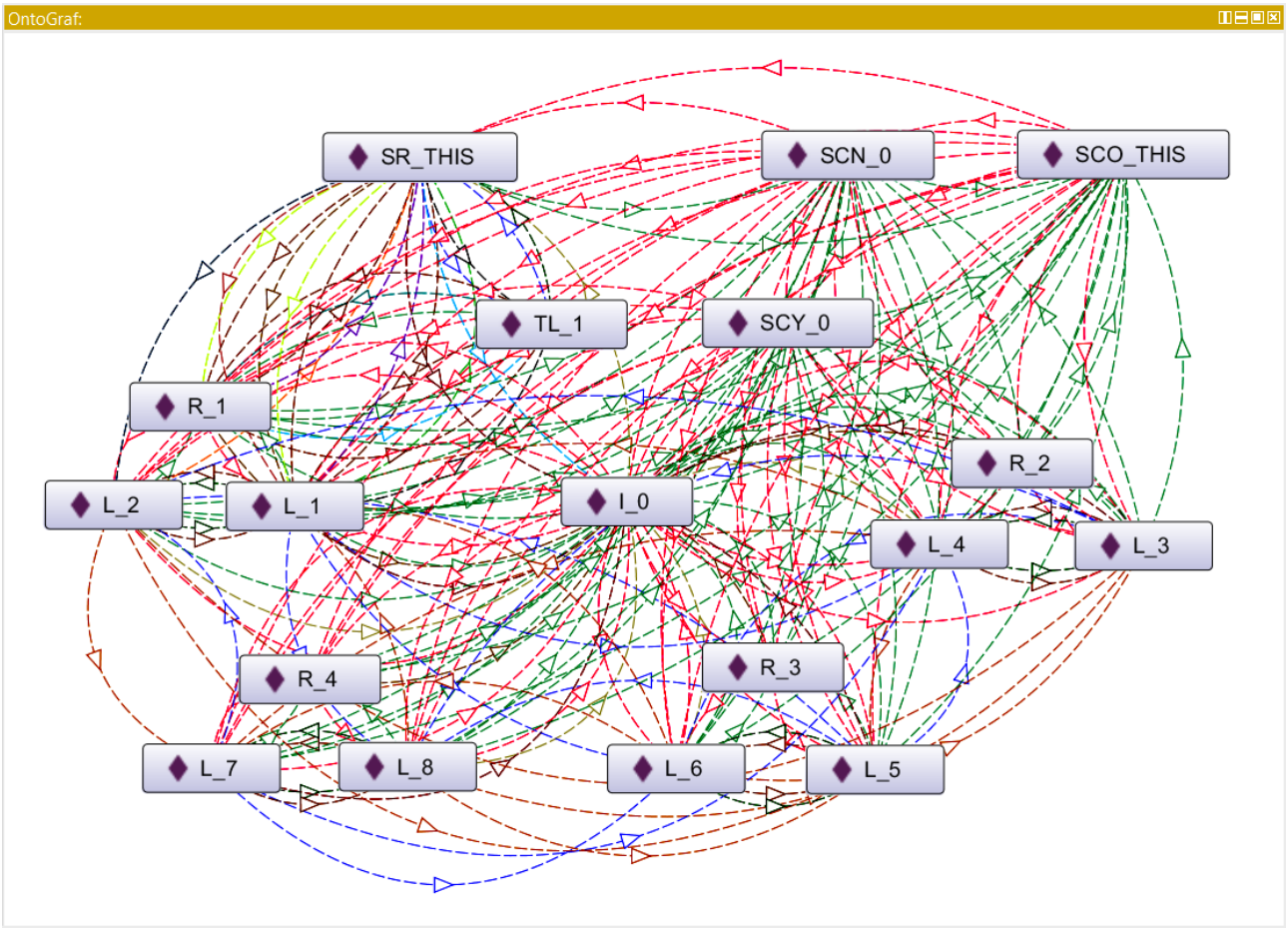


Figure 4-6: Exemplary intersection scenario knowledge base after inferencing. Because of the specified relations and axioms in the ontology, an inference engine is able to derive further information. Hence, additional relations between the individuals in the knowledge base are asserted.

plugin, the *SWRL Tab*. Appendix A.3 lists the entire set of rules defined in the context of the developed ontology. As presented in the following, the expressiveness of the ontology is increased by the addition of these semantic axioms.

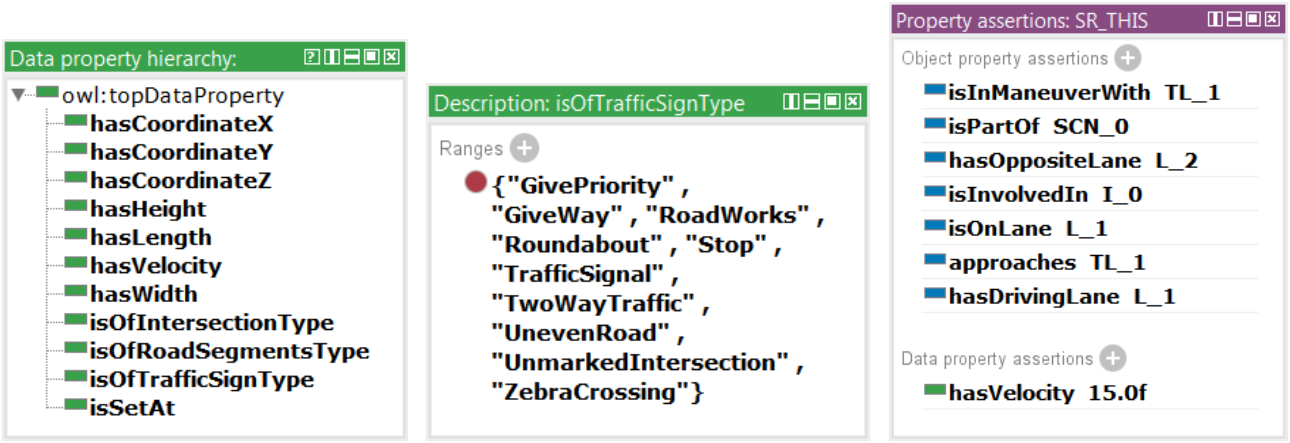
As described in section 2.2.1, ontologies show their true strength when their reasoning resp. inference capabilities are exploited. Inferencing allows to draw semantic conclusions about previously undefined class properties and relations from given facts. In this context, inferencing is used to preserve unspecified relations between instantiated individuals that can be derived from the knowledge stored in the ontology. To illustrate this process, the previous example from Figure 4-5 is considered again. When the knowledge base represented therein is examined by an inference engine, further relations between the modeled individuals are derived. The result is a number of inferred relationships. Figure 4-6 shows the same knowledge base, but after inferencing.

Inferencing fulfills three central tasks in the context of this work's approach. As the example illustrates, inferencing enhances the resulting knowledge bases and ultimately represents a multiple of the previously specified relationships. In addition, inferencing enables a kind of unification of the developed knowledge bases in this way. The fact that all defined rules are applied to all individuals ensures that the resulting knowledge bases have the same expressiveness and are therefore compatible. In addition, modeling errors and rule violations are reported by the inference engine. This enables the correction of such errors.

Therefore, the aforementioned modeling possibilities allow the creation of different scenario knowledge bases. On closer inspection though, it becomes apparent that only functional dependencies are specified in accordance with the definitions of Steimle et al.[100]. As elaborated in Section 3.2.1, however, the representation of logical and concrete scenarios is also intended. For this purpose, the ontology is further extended. In order to be able to represent concrete values or value ranges, *properties* are specified in addition to classes and relations.

Figure 4-7 shows the properties defined in the context of the developed ontology. It is also depicted how the description of an individual can be concretized by these properties. The property concepts listed in Figure 4-7 (a) mostly model classical physical properties of real objects. Examples are values for height, length and width as well as coordinates and velocities. These are assigned explicit value types such as int, float or string. In addition, properties with finite value sets are specified. In Figure 4-7 (b), for example, the description of the isOfTrafficSignType property is displayed with its value set. In this case it is modeled that instances with this property always represent exactly one of the listed traffic sign types. Figure 4-7 (c) shows how properties are included into the description of individuals. In this example the self-representation SR_THIS is assigned a longitudinal velocity with the value "15.0f". The "f" indicates that this is a value of type float. The X, Y and Z descriptions in the example properties refer to an arbitrarily defined, fix world coordinate system while velocities are formulated in meters per second.

[100] Steimle, M. et al.: Terminologie für den szenarienbasierten Testansatz (2018).



(a) Property list (b) Property description (c) Property assertion

Figure 4-7: Definition and use of properties in the developed ontology. In Figure (a) several property concepts included in the ontology are presented. The value range of the isOfTrafficSignType property is depicted in Figure (b) and an exemplary assertion of the hasVelocity property to an individual is shown in Figure (c).

The knowledge bases created in this way are eventually utilized to represent the operational scope of the overall software framework. A knowledge base is thereby understood as a scenario and the combination of several knowledge bases is interpreted as a scenario catalog. An interface called *OWLready2* [101] enables the integration of these digital artifacts into Python.

[101] Lamy, J. B.: Owlready: Ontology-oriented Programming in Python (2017).

4.3 System Architecture

In the context of the implementation, the system's architecture is represented as a task-chain-pattern skill graph. This type of modeling is introduced and explained in Section 3.3. Task-chain-pattern skill graphs are composed of nodes as well as edges and always depict directed acyclic graphs. Therefore, it is desired to be able to store such a graph as a machine-readable artifact in order to integrate it into the software framework. This is realized by the Python library *NetworkX*[102]. *NetworkX* not only offers functionalities for modeling directed graphs, but also provides algorithms for finding particular structures in graphs, such as subgraphs or shortest paths. The individual nodes and edges of the graphs can also be more precisely specified by appending further information during modeling. In addition, *NetworkX* can be used to generate random graphs and export any resulting graphs as an images.

Task-chain-pattern skill graphs are a special form of skill graphs. They represent a system's architecture as the union of a multitude of skills and their dependencies on each other. In the practical application of the presented approach, it is necessary to bring in expert knowledge in order to map the system's architecture correctly. In the following the modeling of such a skill by using *NetworkX* is explained on the basis of a fictitious example. It is modeled in which way a system detects traffic lights and reacts to them by adjusting its speed. The example was modeled on the basis of publications by Ziegler et al.[103] and Reschka et al.[104] and is presented in Figure 4-8. Ziegler et al. describe how an autonomous vehicle recognizes and classifies traffic lights. These explanations are used as a starting point for the modeling process. Reschka et al., on the other hand, offer examples of skills in relation to an ACC system. Some skills of their proposals are adapted to model the example.

Figure 4-8 shows the exemplary task-chain-pattern skill graph developed in this context. The illustration does not represent the entire architecture of the system, but only one of its main skills. In this example the main skill Handle Traffic Light is shown, which in turn is composed of different skills and also has dependencies to sensors and actuators of the system. The graph only serves to illustrate the task-chain-pattern skill graph implementation. Therefore, no claim to the completeness of the modeling is made. For a better understanding of the following explanation, it is referred to Section 3.3 and the therein described modeling guidelines. The modeling guidelines have been followed when creating the exemplary task-chain-pattern skill graph. The entry point marks the top of the graph. From this point it is possible to reach all nodes belonging to the main skill via their respective dependencies.

Starting from the ends of the graph, i.e. the sensors and actuators, the illustrated system capabilities become clear. The sensors are needed to realize certain perception and comprehension skills. The LongRangeRadar sensor enables the ComputeRadarReflexions skill and the StereoCamera detects so-called stixels in the ComputeStixels skill. The WideAngleCamera and StereoCamera sensors are both needed to classify image sections through the ClassifyImageSegment skill. By the results of the perception skills, it is possible to detect various static objects.

[102] NetworkX developers: NetworkX – Software for Complex Networks (2018).
[103] Ziegler, J. et al.: Making Bertha Drive (2014).
[104] Reschka, A. et al.: Ability and Skill Graphs (2015).

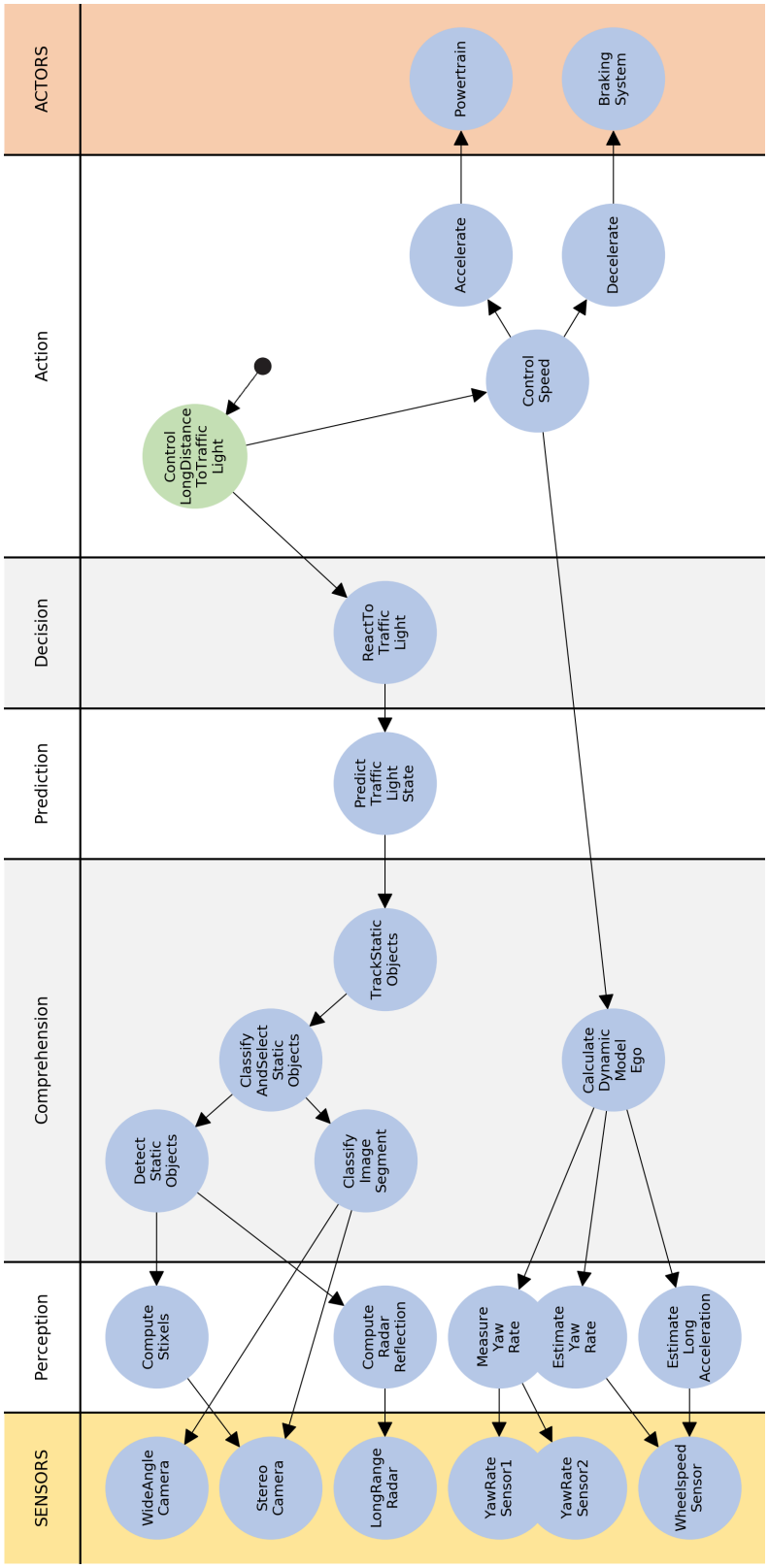


Figure 4-8: Task-chain-pattern skill graph representation of an exemplary, fictitious Handle Traffic Light skill. The graph's structure is based on publications of Ziegler et al.[105] and Reschka et al.[106].

[105] Ziegler, J. et al.: Making Bertha Drive (2014).
[106] Reschka, A. et al.: Ability and Skill Graphs (2015).

This is modeled by the DetectStaticObjects skill, followed by the ClassifyAndSelectStaticObjects skill. The latter depends on both previous skills. All detected objects are combined with the classified image segments and the selection of certain objects of interest, in this case traffic lights, is realized. The following TrackStaticObjects skill tracks the previously selected objects. Then, based on this tracking, a prediction of the state of the traffic light takes place, modeled by the PredictTrafficLightState skill. Subsequently, the previous steps of the ReactToTrafficLight skill are used to react to the traffic lights, i.e. a decision is made. Only the ControlLongDistanceToTrafficLight is connected to this chain which models the regulation of the longitudinal distance between the system and a traffic light object. It thus represents an action skill that expresses a behavior of the system that can be observed from the outside. This skill is called a top skill and is directly connected to the entry point.

The ControlSpeed skill, which is related to the top skill, also models an action and is necessary to control the distance to traffic lights. It is conditioned by three additional skills. On the one hand, a dynamic model of the system resp. autonomous vehicle is needed to control distances to other objects. This is modeled by the CalculateDynamicModelEgo skill, which in turn depends on several perception skills and sensors. On the other hand, a system (at least in this simple model) for controlling distances to other objects must be able to influence its own velocity. This is expressed by the Accelerate and Decelerate skills, which in turn are directly dependent on the Powertrain and BrakingSystem actuators.

The dependencies of the main skill HandleTrafficLight to the associated sensors and actuators as well as the dependencies between the chronologically arranged skill nodes become apparent. To further increase the expressiveness of task-chain-pattern skill graphs, additional information is included in the modeling process. *NetworkX* offers the possibility to add further information to the nodes and edges. In the previous example, performance parameters of the sensors and actuators are specified in this way. Alternatively, the modeling of skill node latencies or the weighting of edges for the realization of more detailed dependencies between skills is conceivable.

For the example, the performance parameters of the WideAngleCamera, StereoCamera and LongRangeRadar sensors are specified based on the sensor set of in the publication of Ziegler et al.[107]. Each of the three sensor nodes then contains information about its sensor range and horizontal field of view. Table 4-1 shows the adopted sensor parameters. In addition, a maximum brake deceleration is defined in the BrakingSystem node actor. To keep the example simple, the deceleration is modeled invariably and independently of external influences with $8 m/s^2$. It is also listed in Table 4-1. The relevance of the integration of such performance parameters for the final derivation of dependency chains is presented in the following Section 4.4.

The overall task-chain-pattern skill graph modeled by *NetworkX* finally represents the architecture of the system. Because *NetworkX* is a Python library, it is possible to integrate the resulting model directly into the developed software framework. The chain derivation engine explained subsequently contains this representation of the system's architecture as an information source and utilizes the algorithms offered by *NetworkX* for finding paths between main skills and system components.

[107] Ziegler, J. et al.: Making Bertha Drive (2014).

Table 4-1: Performance parameters of the WideAngleCamera, StereoCamera and LongRangeRadar sensor nodes and the BrakingSystem actor node in the context of the exemplary task-chain-pattern skill graph. The values are based on Ziegler et al.[108].

	WideAngle Camera	Stereo Camera	LongRange Radar	Braking System
Range R	130m	80m	200m	–
Horizontal field of view α_{H}	90°	44°	18°	–
Maximum deceleration $d_{\max}$	–	–	–	8m/s^2

[108] Ziegler, J. et al.: Making Bertha Drive (2014), based on Fig. 3, p. 4.

4.4 Chain Derivation Engine

The chain derivation engine is considered to be the algorithmic core of the implemented software framework. Its task is to use the operational scope and the system's architecture as incoming information sources to automatically derive the intended dependency chains between particular circumstances in scenarios and thereto associated architectural elements. To enable this, the chain derivation engine applies its semantic rules and utilizes the inherent arithmetic. In the following, this construct will be clarified with reference to previous sections as well as with an example. First, the structure of an exemplary semantic rule is presented, followed by an arithmetic function, which is necessary for the rule's application. Based on this, the algorithmic of the chain derivation engine and resulting analysis functionalities are presented.

As outlined in Section 3.4.1, the semantic rules are formulated using the RDF query language SPARQL. The functionalities of SPARQL make it possible to automatically recognize and extract certain circumstances in the modeled knowledge bases of the operational scope. In addition, further parameters are passed during the initialization of a rule object. For the example, following simple rule is generated:

```
Rule('124',
     """SELECT ?SR WHERE {
        ?SR rdf:type <#SelfRepresentation> .
        ?TL rdf:type <#TrafficLight> .
        ?SR <#approaches> ?TL .
        ?SR <#hasVelocity> ?v .
     }""",
     "HandleTrafficLight",
     {'R': ['v', 'dmax']})
```

The rule is initialized by four different parameters. First, a number is assigned to the rule, so that it can be classified in a rule catalog within the framework. In the example the number `'124'` is chosen randomly. This is followed by the inclusion of the SPARQL query which expresses the semantic context addressed by the rule. In this example, it is queried whether an individual `SR` of type `SelfRepresentation` exists, that approaches an individual `TL` of type `TrafficLight`. It also checks if the individual `SR` has the property `hasVelocity`. If this correlation is found in the operational scope, the individual `SR` is selected. Thus, cases are searched in which the system approaches a traffic light with an arbitrary speed and the self-representation individual of the corresponding knowledge base is extracted. The knowledge base discussed in Figure 4-5 and Figure 4-7 illustrates such a case. Apart from that, the parameter `"HandleTrafficLight"` is passed during the initialization of the rule. As a result, reference is made to the system's architecture main skill which is required by the queried context. In the example, a purely functional dependency, stating that when an autonomous vehicle approaches a traffic light, it must then be able to handle it, is modeled. Finally, a set of

performance parameters is passed at the initialization of the rule to represent a logical or concrete dependency beyond the functional dependency. In this case, on the one hand, these are the performance parameters $d_{\max}$ ('dmax') and R ('R'), presented in the previous Section 4.3. On the other hand, the list includes a parameter 'v' as well. Here, the latter refers to the longitudinal velocity of the individual SR. In this way it is modeled that the context addressed by the rule explicitly depends on the two performance parameters mentioned as well as on the velocity of the self-representation modeled in the knowledge base.

This rule hence allows the chain derivation engine to derive dependencies between the addressed context, the named main skill, and system components that contain the listed performance parameters. However, how are the performance parameters included in the derivation exactly? To answer this question, the following example will be further elaborated. For this purpose a metric constraint is defined, which describes the example more concretely.

The following condition is assumed: If an autonomous vehicle approaches a red traffic light with a certain velocity, it must be ensured that the range of its sensors is sufficient to initiate braking in time. In this context, 'in time' means that the minimum braking distance $D_{\min}$ of the vehicle must not exceed the range R of the sensors. For simplification purposes of this example, the minimum braking distance $D_{\min}$ is assumed to be the quotient of the squared longitudinal velocity v and the double maximum deceleration $d_{\max}$ of the system. Using the defined performance parameters, the formulated metric constraint is formulated by

$$R \overset{!}{>} D_{\min} = \frac{v^2}{2d_{\max}}. \tag{4-1}$$

It is further added as a metric function to the Arithmetic class used by the chain derivation engine. If the previously introduced rule is utilized, the arithmetic function is automatically included in the derivation of arising dependency chains, resulting in a concrete requirement for certain components of the system.

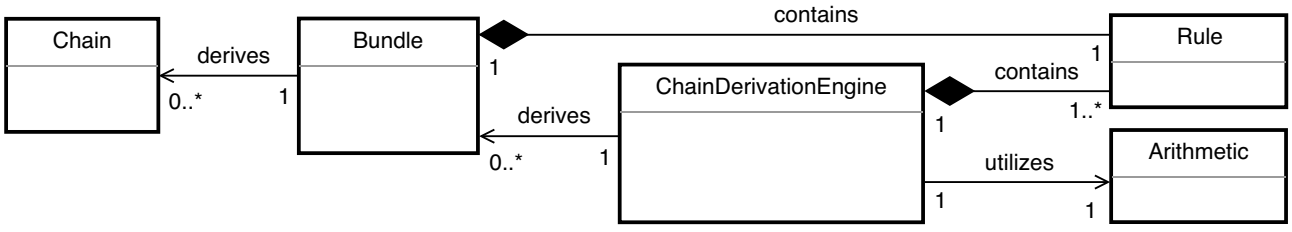


Figure 4-9: Simplified UML class diagram of the implemented software framework in regard to the dependency chain derivation process. The ChainDerivationEngine derives instances of the Bundle class from which subsequently instances of the Chain class are derived. Each Bundle contains exactly one Rule and serves to structure the derivation process.

Subsequently, it is examined more closely in which order the chain derivation engine processes its information sources and how the intended dependency chains are eventually derived. As explained previously, here, the information sources are the digital artifacts of the operational scope and system's architecture. From this, the chain derivation engine does not directly derive the dependency chains, but uses certain intermediary products. This serves to structure the software framework. As shown in Figure 4-9, there are so-called Bundle instances on the way to the derived Chain instances. These bundles represent the mentioned intermediary products. To illustrate how a bundle is created resp. what it represents, the following pseudo code is examined:

```
for SCO in OperationalScope:
   for RULE in RuleCatalog:
      Occurences = SCO.query_owlready(RULE)
      for OCCURENCE in Occurences:
         Obj = OCCURENCE[0]
         newBundle = Bundle(SCO, RULE, Obj)
```

The process symbolizes a part of the implemented program structure. At the highest level, the scenarios contained in the `OperationalScope` are iterated. As explained before, the scenarios are represented by knowledge bases which again are based on the developed ontology. Then, all rules of the chain derivation engine's `RuleCatalog` are traversed. The `RuleCatalog` is the collection of all specified semantic rules. Here, the respective rule is applied to the current scenario. For this purpose the functionalities of the Python library RDFLib[109] are utilized. In this way, all correlations addressed by the rule are extracted from the scenario and stored in the `Occurences` list. Subsequently, these `Occurences` are iterated. A defined object of interest is extracted from each single `OCCURENCE`. Finally, an instance of the `Bundle` class is created. Not only the respective scenario and the current rule but also the extracted object are passed thereby. While a `Bundle` unites several pieces of information, it does not yet represent holistic dependencies in this context. Rather, the information contained describes an explicit incident in a specific scenario addressed by a single semantic rule.

Up to this point, no information about the system's architecture is included in the process. Yet, this occurs in the following step, in which the final chains are derived from each bundle. The following pseudo code excerpt illustrates the procedure:

```
for BUNDLE in Bundles:
   BUNDLE.derive_Chains(SystemArchitecture)
```

Here, the `Bundles` resulting from the previous step are processed individually. In each iteration the `derive_Chains(...)` function is called and an instance of the `SystemArchitecture` class is passed. This allows the bundles to be repeatedly charged with varying architectures. For instance, this enables a clear separation of the results when two different system architectures are to be compared. With regard to the concrete chain derivation process, the presentation of more specific program sec-

[109] Grimnes, G. A.: RDFLib (2018).

tions is deliberately omitted in order to keep the explanations within an appropriate boundary. Instead, the entire chain derivation process is subsequently illustrated by an overarching example.

Following the examples presented in the previous sections are considered collectively. First, the scenario presented in Figure 4-5 is assumed to have a self-representation individual with the properties according to Figure 4-7 (c). Thus, the operational scope is modeled based on this single scenario and the self-representation in it approaches a traffic light with a longitudinal velocity of 40 m/s. Secondly, the example is based on a system whose architecture is only represented by the HandleTrafficLight main skill displayed in Figure 4-8. The sensors and actuators of this system architecture are described by the performance parameters shown in Table 4-1. Furthermore, the semantic rule discussed on Page 51 is added to the rule catalog of the involved chain derivation engine. The corresponding arithmetic function is also included.

In this way, all information required for the automatic derivation of dependency chains is provided. Eventually an instance of the Analysis class summarizes all this information in the software framework. As shown in Appendix A-1, this class contains, among other things, an `about(...)` function and a `visualize(...)` function. If the first function is used, a textual printout results, which provides information about the quantity and characteristics of the derived bundles and chains. The second function allows a more expressive visualization of the derived dependency chains. Subsequently, the functions' outputs resulting from the collective example are presented.

In regard to the example, the `about(...)` function returns the following output:

```
#1
Rule:          #124
Scenario:      SC01
Main skill:    HandleTrafficLight
Object:        SC01.SR_THIS
Metric(s):     {'R': ['v', 'dmax']}
  [False, [100.0], [80]]
  [False, [100.0], [80]]
  [True, [100.0], [130]]
  [True, [100.0], [200]]

TOTAL
  4 derived chains
  2 critical
  2 uncritical
```

Hereby, all derived bundles and their key information is displayed textually. Since the context addressed by the semantic rule occurs only once in the examined scenario, only a single bundle results. Besides the inherited information about `Rule` number, `Scenario` name, `Main Skill`, `Object` and the performance `Metric(s)` a short list concerning the dependency chains derived from the bundle

is shown. The color coding and the identifiers `True` resp. `False` indicate whether the chains are `uncritical` or `critical`. A chain is considered uncritical if the requirements expressed by it are met by the modeled system architecture. In turn, a chain is regarded as critical if the opposite is the case. The function also provides an overall review of the dependency chains derived from all bundles. The example shows that four dependency chains are derived in total. Two of these chains are critical, while two are uncritical.

However, this printout does not yet allow to draw far-reaching conclusions. For example, it does not show which system components are included in the chain, nor why exactly a chain is classified as uncritical or critical. To support more dedicated analysis functionalities, the `visualize(...)` function of the Analysis instance is used. It allows full, detailed visualizations of the automatically derived dependency chains. Additionally, the function provides a classification explanation in natural language for each chain.

Figure 4-10 shows the output of the analysis function in reference to the example. Here, the four automatically derived dependency chains are presented graphically. The affected system components, the main skill, the semantic rule, the scenario as well as the individual extracted from the scenario are plotted. For the sake of clarity, however, not all skill nodes are listed. Instead, they are replaced by a substitute node with a "..." label. The complete dependency chains are presented in Appendix A.4 including the underlying HandleTrafficLight main skill. Due to the simplicity of the example, the four chains in Figure 4-10 differ only by the therein linked sensors. In fact, the first two chains appear identical in this simplified representation. When inspecting the illustrations in Appendix A.4, however, it becomes clear how the chains differ from each other. The difference lies in the respective path of the task-chain-pattern skill graph. Based on the defined performance parameters, the chain derivation engine automatically derives all feasible paths from the entry point of the main skill to corresponding system components. Since, in this example, exactly two paths from the entry point to the StereoCamera node are feasible, the first two effect chains result. Based on the performance parameters of the connected system components and the properties of the individual modeled in the knowledge base, the chain derivation engine applies the stored arithmetic function. Finally, the result of the calculation determines whether a chain is classified as uncritical or critical. In the example, this means that a chain is only uncritical if Equation 4-1 is true with the setting of the respective parameter values. However, if the condition of the function is not fulfilled, the chain is classified as critical. In addition to the color coding (uncritical = green, critical = red), the result is also explained in natural language. In the case of the first two chains, for example, it is explained that the StereoCamera facilitates a range of $R = 80\,\mathrm{m}$. But, due to the parameters $v = 40\,\mathrm{m/s}$ and $d_{\max} = 8\,\mathrm{m/s^2}$, the rule requires a greater sensor range resulting from the calculated minimum braking distance $D_{\min} = 100\,\mathrm{m}$. Consequently, the condition of the rule's metric constraint is not fulfilled. For the two bottom chains in Figure 4-10 another result is obtained. Here, the condition is fulfilled due to the higher sensor ranges of the WideAngleCamera and the LongRangeRadar. Accordingly, the chains are classified as uncritical.

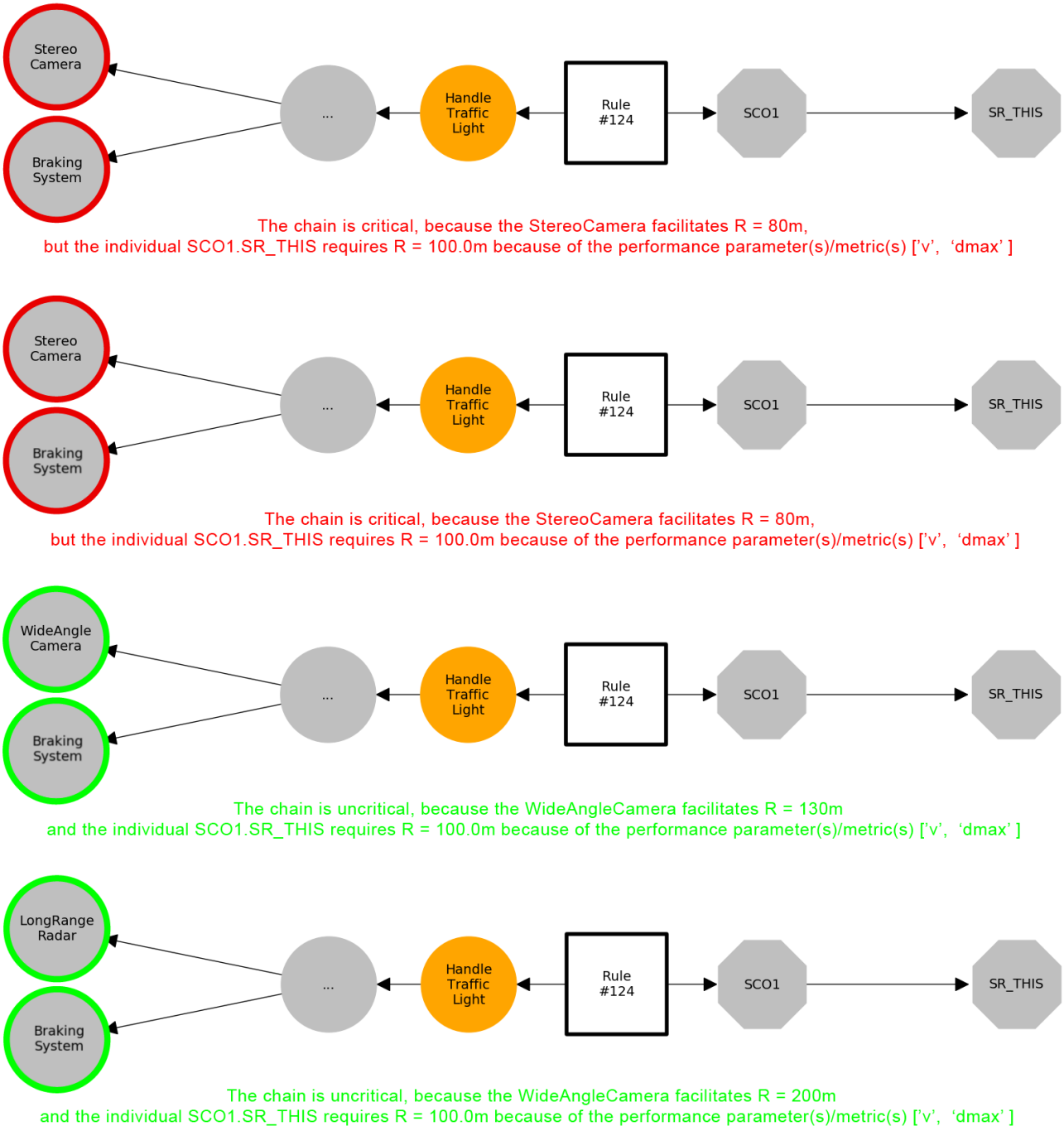


Figure 4-10: Four automatically derived dependency chains in regard to a holistic example. The shown visualization is a shortened output of the implemented `visualize(...)` function. The affected system components, the main skill, the semantic rule, the scenario as well as the respective individual extracted from the scenario are depicted. Whether and why a chain is critical or not is expressed by color (red/green) and explained in natural language. The corresponding, full dependency chains are presented in Appendix A.4.

The holistic example demonstrates the extent to which the implemented software framework automatically identifies and extracts dependency chains based on the previously modeled digital artifacts resp. information sources. Furthermore, it is clarified how requirements for certain system components are derived. Through the criticality classification of the resulting dependency chains, additionally, con-

flicts between the intended operational scope and the system's architecture are revealed. Based on the implemented methods it becomes feasible to elaborate further analysis functionalities for the support of system design processes. The potentials and challenges of the approach are discussed in Chapter 6.

5 Evaluation

In this chapter, the presented approach is evaluated using the implemented software framework functionalities. The approach is examined with regard to scalability, or more precisely load scalability. Initially, the understanding of the load scalability metric in the context of this work is explained. Then several parameters are introduced that aim to enable a quantitative analysis. Finally, the conducted experiments and their discussion are presented.

5.1 Load Scalability

The approach presented in this work is intended to support the development of systems for automated driving. As explained in Section 1.1, these systems have challenging high complexities. In addition, todays developers aim to enable autonomous vehicles to handle increasing amounts of various scenarios in the future. When applied to the proposed approach, this means that the representation of the system architecture and also the developed ontology must allow the description of much more complex relationships than the examples in Chapter 4. Furthermore, this requires the development of a rich catalog of semantic rules to enable the chain derivation engine to derive complex dependency chains. As a result, the amount and complexity of information sources of the software framework rises. In other words: the work load increases. In order to be able to assess whether the presented approach meets this claim, it is charged with different work loads in Section 5.2. For this purpose, the implemented software framework is used to derive dependency chains in regard to differently complex variations of the OperationalScope, the RuleCatalog and the SystemArchitecture. To allow a quantified investigation, the resulting computation times and the number of derived dependency chains are compared. Building on this, the load scalability of the approach is discussed.

But what exactly is understood as load scalability in the context of this work? As Hill[110a] states, general scalability is an often used attribute in the description of software-based systems, but "*has no generally-accepted definition*"[110b]. He also explains that the term is only useful in a technical sense if a strict definition is being offered.[110b] For this reason, a most explicit definition of the load scalability term is elaborated. In general, the scalability of a system can be manifested in various ways.[111] Scientists use a variety of scalability metrics, e.g. speed, efficiency, size, application, generation and heterogeneous scalability.[111] The term workload or load scalability is used as well. Bondi[112a] understands the load scalability term as follows:

> *"A System has load scalability if it has the ability to function gracefully [...] at light, moderate, or heavy loads [...]."* [112b]

In the context of this evaluation, his understanding of the attribute is used as an orientation. Nevertheless, the definition is further detailed by using technical terms to make the condition more explicit.

[110] Hill, M. D.: What is Scalability? (1990), a: – ; b: pp. 1-2.
[111] El-Rewini, H.; Abd-El-Barr, M.: Advanced Computer Architecture Programming (2004), pp. 63-66.
[112] Bondi, A. B.: Characteristics of Scalability and Their Impact on Performance (2000), a: – ; b: p. 2.

On the one hand, it is determined that the graceful functioning of a system is measured by its concrete processing times. On the other hand, a linear relationship between the processing times and the work loads is implied in order to refer to the consequences of arbitrarily increasing work loads. Similarly, linear correlations are also implied in the description of the general scalability of systems.[113] If the processing times of a system would not increase linearly in relation to its workload, but exponentially, for example, the system would consequently not be considered as load scalable. Yet, if there is a linear correlation, it is assumed that the increase in processing time might be successfully counteracted by increasing the systems resources.

Load scalability: A System has load scalability resp. is load scalable if its processing times do only increase linearly in relation to the magnitude of applied work loads.

The grasp of a work load's magnitude still leaves room for interpretation and is therefore further concretized. As previously indicated, the various sources of information or the software framework's internal artifacts (OperationalScope, SystemArchitecture and RuleCatalog) serve as the work load for this evaluation. The goal is to describe their magnitude with explicitly defined parameters. Three parameters are identified, which are utilized to vary the work load magnitude. These are explained and quantified in the following.

1. **Complexity of a OperationalScope**

   Using the developed ontology, differently detailed scenario presentations can be modeled as knowledge bases. The merger of several such scenarios forms the OperationalScope artifact. Consequently, the complexity of the OperationalScope depends on two subdivided parameters. It is possible to vary both the count and the level of detail resp. complexity of the scenario representations.

   The scenario count of a OperationalScope OS is following indicated by n_s and describes the concrete number of included knowledge bases. The complexity of a scenario s_i in turn is characterized by $\Lambda(s_\mathrm{i})$, but cannot be measured trivially. A quantification is therefore established. This is based on descriptions of Schuldt[114a] and Yang et al.[115]. Schuldt addresses this measure by discussing various related publications. He demonstrates that the complexity of virtual environments like the here used scenarios depends on a multitude of dimensions.[114b] Two of these dimensions are the number of elements as well as their interconnectedness. Yang et al.[115] investigate the complexity of ontologies in general and propose a metric that depends on the number of classes and the count of relations. Since the authors' descriptions overlap and corresponding values are part of the ontology based scenario presentations, these two dimensions are assumed to measure the complexity $\Lambda(s_i)$ of a scenario s_i. In concrete terms, this means that the scenario complexity $\Lambda(s_i)$ is defined as the sum of all instantiated classes resp. individuals and relations in the respective knowledge base.

[113] El-Rewini, H.; Abd-El-Barr, M.: Advanced Computer Architecture Programming (2004), p. 63.
[114] Schuldt, F.: Diss., Methodisches Testen von automatisierten Fahrfunktionen (2017), a: –; b: pp. 19-22.
[115] Yang, Z. et al.: Evaluation Metrics for Ontology Complexity (2006).

In this manner, the complexity of an OperationalScope *OS* is measurable. Since an OperationalScope represents a simple concatenation of separate knowledge bases and these are also treated strictly separated from each other in the chain derivation process, the complexity of an OperationalScope *OS* is defined as

$$\Lambda(OS) = \sum_{i=0}^{n_\mathrm{s}} \Lambda(s_i)\,. \tag{5-1}$$

2. Complexity of a RuleCatalog

As key components of the chain derivation process, the semantic rules must also be included in the analysis of the load scalability of the approach. The collection of multiple such rules contained in a ChainDerivationEngine is called a RuleCatalog. The complexity of a RuleCatalog also depends on two subdivided parameters. Again, both the number and complexity of the rules are variable.

The amount of rules in a RuleCatalog *RC* is denoted by n_r and indicates how many instances of the Rule class are contained in a corresponding ChainDerivationEngine. Similar to scenarios, the complexity of a Rule r_i is described by $\Lambda(r_i)$. However, no trivial dimensioning of complexity is possible here either, which is why this metric is also quantified. The main part of the semantic rules are the RDF-based SPARQL queries. SPARQL is a graph-pattern matching query language and has dedicated querying capabilities.[116] Pérez et al.[117a] explain these querying capabilities and investigate the semantics as well as the complexity of SPARQL queries. They find that although different operator types within a query have differing degrees of influence on its complexity, the size of the requested pattern always influences its complexity.[117b] A query pattern in SPARQL is again composed of a number of RDF tripels. Based on this knowledge, the complexity $\Lambda(r_i)$ of a semantic rule r_i in the context of this work is measured by the number of RDF tripels formulated in an instance of the Rule class.

In this way, again, the complexity of the merger, the RuleCatalog *RC*, is described explicitly. Since the individual rules in the chain derivation process are called sequentially, i.e. separately, they do not influence each other. Hence, the complexity of the RuleCatalog *RC* is defined as

$$\Lambda(RC) = \sum_{i=0}^{n_\mathrm{r}} \Lambda(r_i)\,. \tag{5-2}$$

3. Complexity of a SystemArchitecture

Another data source of the software framework is the SystemArchitecture represented by a directed acyclic graph. Depending on the level of detail of the modeled task-chain-pattern skills of the system, the number of nodes and edges in the graph increases. However, the complexity of

[116] Prud'hommeaux, E.; Seaborne, A.: SPARQL Query Language for RDF (2008).
[117] Pérez, J. et al.: Semantics and Complexity of SPARQL (2006), a: – ; b: pp. 32-37.

a graph is not simply described by the number of these concepts in the graph. Instead, the field of graph theory offers a multitude of complexity metrics. It is also known, however, that the choice of the appropriate metric always depends on the nature and application of the respective graph. Since no dedicated metric seems to exist for the type of graph used in the context of this work, a new complexity metric is developed after consideration of generally accepted metrics. The aim is to define the complexity of a system architecture represented as graphs respecting its use in the course of the chain derivation process.

In the presented approach an instance of the SystemArchitecture class is utilized to derive any paths between a main skill and certain system components. The more such paths are enabled by the structure of the graph, the more dependency chains result. Based on this knowledge, the intended complexity metric is derived. It is defined that the complexity $\Lambda(SA)$ of a SystemArchitecture corresponds to the average number of possible paths between a skill and a system component in its inherent graph. With the definition made the complexity of a SystemArchitecture *SA* is quantifiable.

It is assumed that the process times of the implemented approach are influenced by these three work loads and their complexities. In order to examine which exact dependencies are present, the experiments described in the following section are carried out and evaluated. Finally, the load scalability of the approach is discussed.

5.2 Experiments and Survey

Based on the defined evaluation metrics, experiments are carried out to investigate the load scalability of the suggested approach. For this purpose fictitious scenarios, rules and architectures of different complexity are modeled in order to map different levels of the complexities $\Lambda(OS)$, $\Lambda(RC)$ and $\Lambda(SA)$. In the beginning it is presented how this modeling is realized and which assumptions are being met. Subsequently, the influence of the individual complexities on the processing times of the software framework is considered. Based on the results, the load scalability of this work's approach is discussed.

In order to model the operational scope *OS* artifact, comparable with the example of Figure 4-5 in Section 4.2, a knowledge base is created based on the developed ontology. This knowledge base is designed in different degrees of detail. Since there is no direct possibility for an automatic generation of such knowledge bases, it is designed by hand. This means that different numbers of individuals are specified and the amount of relations between these individuals is varied as well. As stated previously, the complexity $\Lambda(s)$ of a scenario s is defined as the sum of all instantiated classes resp. all individuals and their relations in the respective knowledge base. Thus, differently complex scenarios result. In total, seven variations with complexities ranging from $\Lambda(s) = 56$ to $\Lambda(s) = 992$ are modeled. Through differently sized collections of these variations, different complexities $\Lambda(OS)$ of a OperationalScope *OS* are mapped in the experiments. The complexities $\Lambda(s)$ of the variations are listed in Table 5-1.

Table 5-1: Complexities $\Lambda(s)$ of manually created knowledge base resp. scenario variations. The variations are utilized in following experiments to map different complexities $\Lambda(OS)$ of an OperationalScope *OS*.

Scenario variation	1	2	3	4	5	6	7
Complexity $\Lambda(s)$	56	157	257	363	538	738	992

On top of that a modeling of RuleCatalog variations with varying complexity is intended. The creation of corresponding semantic rules is done manually as well. Comparable to the Rule in Section 4.4 a basis SPARQL query is designed. This is then formulated in varying detail by changing the number of RDF tripels. Since the complexity of a Rule was previously defined as the number of tripels, differently complex variations of a Rule *r* result. A total of seven variations of the Rule *r* are selected. The variations feature complexities from $\Lambda(r) = 3$ to $\Lambda(r) = 45$. Through differently sized collections of these variations, different complexities $\Lambda(RC)$ of a RuleCatalog *RC* are mapped in the experiments. The complexities $\Lambda(r)$ of the variations are listed in Table 5-2.

Table 5-2: Complexities $\Lambda(r)$ of manually created Rule variations. The variations are utilized in following experiments to map different complexities $\Lambda(RC)$ of a RuleCatalog *RC*.

Rule variation	1	2	3	4	5	6	7
Complexity $\Lambda(r)$	3	10	17	24	31	38	45

Finally, a fictitious SystemArchitecture *SA* is modeled, too. Since the investigation of preferably complex architectures is intended, a computer-aided generation of the desired SystemArchitecture artifacts is implemented in contrast to the two previous procedures. For this purpose a script is elaborated which automatically generates directed acyclic graphs in compliance with the modeling guidelines from Section 3.3.2. In this way it is possible to create several differently complex models of a system's architecture. In addition, an algorithm is developed which, according to the definition of $\Lambda(SA)$, delivers the complexity of the resulting graphs resp. SystemArchitecture artifacts. For each main skill node the possible paths to all actor and sensor nodes connected to it are counted. Then the arithmetic mean of these counts is formed, from which $\Lambda(SA)$ results. Again, a total of seven variations is selected for the experiments. The different SystemArchitecture artifacts feature complexities from $\Lambda(SA) \approx 0.5$ to $\Lambda(SA) \approx 3.0$. The complexities $\Lambda(SA)$ of the seven variations are listed in Table 5-3.

Table 5-3: Complexities $\Lambda(SA)$ of automatically generated SystemArchitecture variations. The variations are utilized in following experiments to map different complexities of a SystemArchitecture *SA*.

SA variation	1	2	3	4	5	6	7
Complexity $\Lambda(SA)$ (rounded)	0.490	0.508	1.007	1.502	2.005	2.514	3.014

When creating the scenario and Rule variations, the subsequent creation of corresponding OperationalScope and RuleCatalog variations as well as the generation of SystemArchitecture variations, certain assumptions are met. On the one hand an OperationalScope *OS* always consists of a number of scenarios of the same complexity. Thus it always contains several copies of a scenario that is regarded as average. The same applies to the variations of a RuleCatalog *RC*. Each RuleCatalog is modeled as a collection of a plurality of the same average rule. Thus it is possible to vary the respective sub-parameters (number and complexity) evenly and separately. On the other hand, the semantic rules constantly query an occurrence which appears only once in all scenarios, regardless of their complexity. This serves the comparability of the results in the experiments. Simplifications are also used for the automatic generation of the SystemArchitecture variations. Hereby, all generated graphs have the identical number of main skill nodes, sensor nodes and actor nodes. The only difference between them is the interconnectedness of the start and end nodes. This also aims at a better comparability of the results. Since the artifacts are created in coordination with each other, a special constraint regarding the complexities occurs. In this case, the smallest scenario complexity $\Lambda(s)$ must always be at least equal to the largest Rule complexity $\Lambda(r)$. This is because, through its RDF tripels, a SPARQL query can only query as many occurrences in a knowledge base as there are in it. As mentioned before, seven variations of each data source are included in the experiments. This aims to keep the evaluation costs within reasonable limits.

First, the impact of the OperationalScope complexity $\Lambda(OS)$ on the process execution of the implemented software framework is investigated. According to the previously developed definition, $\Lambda(OS)$ is composed of the sum of the complexities of all scenario representations included in the Opera-

tionalScope. Thus there are two setting parameters for varying the complexity $\Lambda(OS)$. Namely these are the scenario complexity $\Lambda(s)$ and the scenario count n_s. Due to the assumptions made, collections of identical, average scenarios are considered here. Therefore, the parameter $\overline{\Lambda}(s)$, which represents the average complexity of the scenarios in the respective OperationalScope, is referred to in regard to the scenario complexity. A variety of OperationalScope artifacts of different complexity is created by a controlled variation of the two parameters. The complexity of the scenarios is varied according to the seven values in Table 5-1. The number of scenarios is also varied in seven incremental steps from $n_s = 1$ to $n_s = 120$. More increments or higher values are not feasible due to limited computing time in the context of this paper.

The software framework is then charged with the resulting OperationalScope artifacts of varying complexity. Through the process, the RuleCatalog and the SystemArchitecture are passed as further input variables, however, are not changed. All measurements are carried out ten times in order to counteract measurement inaccuracies. The investigation of the load scalability focuses on the resulting processing times. In addition, the set of derived dependency chains is analyzed in order to be able to observe further effects. In Figure 5-1 the results are represented by three diagrams. Here the same measurement results are presented from three different points of view in order to improve to recognition of any effects. The first diagram depicts a three-dimensional view. The axes plot the average scenario complexity $\overline{\Lambda}(s)$, the scenario count n_s and the resulting processing times. The measured values are shown in the form of circles. As the attached legend indicates, the diameters of these circles represent the count of derived dependency chains. The lower two diagrams represent two orthogonal parallel projections of the upper diagram. On the left the processing time is plotted versus the scenario count n_s and on the right the processing time is plottet versus the average scenario complexity $\overline{\Lambda}(s)$. Here, the diameters of the circles represent the count of derived dependency chains, too. The measured values connected with lines have identical complexity values (left) or identical count values (right).

It becomes apparent that the processing time increases significantly with increasing scenario count n_s. The correlation between scenario count n_s and processing time is approximately linear in the considered value range. This was to be expected, since the individual scenarios in the chain derivation process are treated sequentially and clearly separately from each other. In addition, it becomes apparent that with increasing scenario count the number of resulting dependency chains also increases. This is explained by the met assumptions. Since each of the scenarios models the same event, the number of chains depends merely on n_s. The impact of the average scenario complexity $\overline{\Lambda}(s)$ on the processing time does not seem to be entirely obvious. In the lower right diagram in Figure 5-1 it is noticeable that there seems to be a general increase in processing times with an increase of $\overline{\Lambda}(s)$ in the considered value range, but this increase is significantly lower than by variation of n_s. This is illustrated by the gradients of the drawn connecting edges in the graph. In addition to the lower gradients, a stair-like pattern is visible. The processing times do not increase evenly with an increase of $\overline{\Lambda}(s)$, but seem to stagnate with certain increment steps (for example in the step from $\overline{\Lambda}(s) = 363$ to $\overline{\Lambda}(s) = 538$). Because of this, the hypothesis is made that the proposed definition for quantifying scenario complexity lacks a distinct association between $\overline{\Lambda}(s)$ and the required processing time.

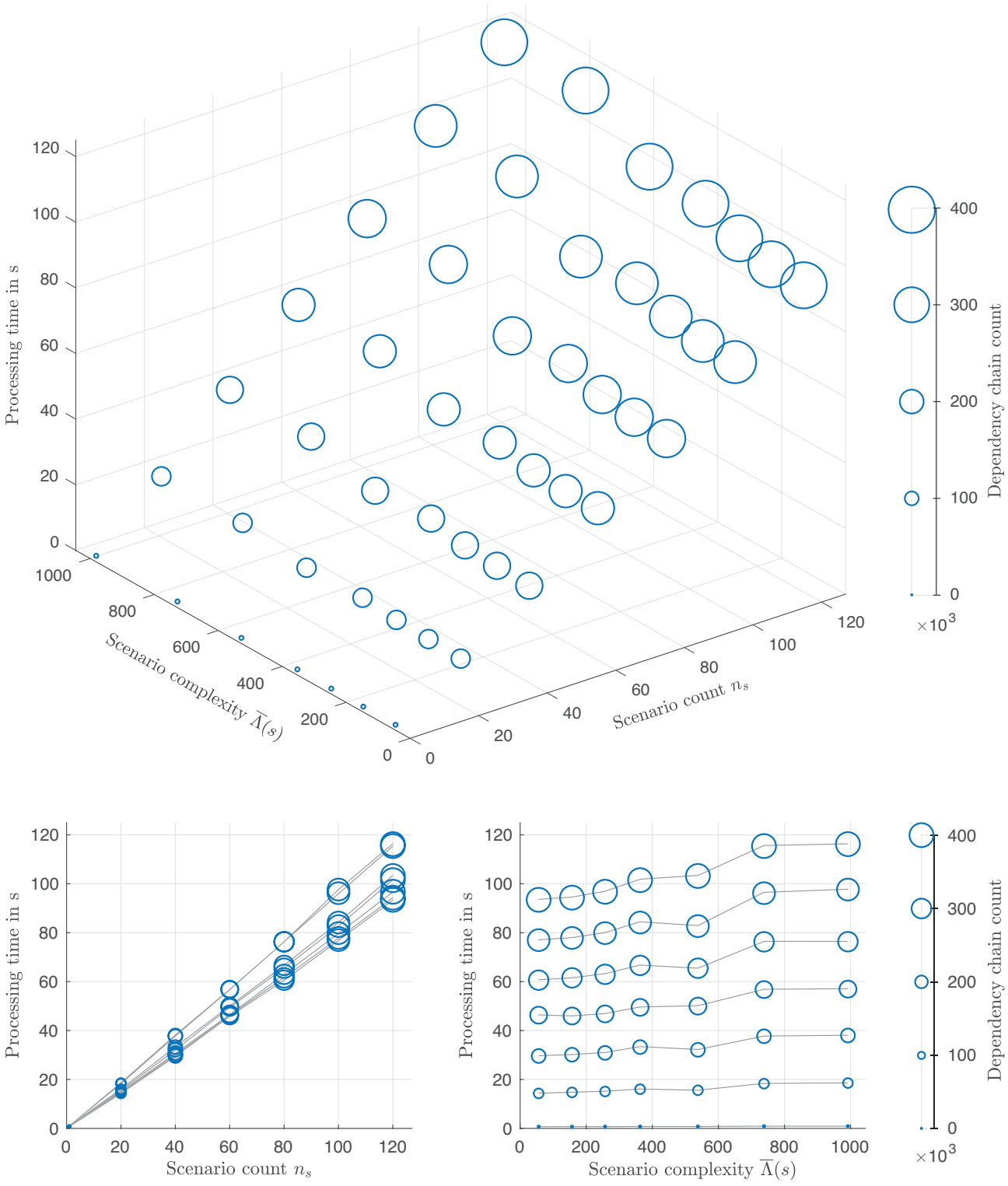


Figure 5-1: Measurement results based on varying scenario complexities $\Lambda(s)$ and scenario counts n_s. By instrumentalization of the implemented software framework measurements of the respective processing times and the count of derived dependency chains are conducted. The three diagrams depict the same measurements from three different perspectives.

Consequently, the mere number of individuals and relations in the knowledge bases does not seem to be the only influence on the processing times. It is also noticeable that the number of derived dependency chains does not seem to depend on the average scenario complexity $\overline{\Lambda}(s)$. Again, this is explained by one of the simplifying assumptions. Since each of the scenarios models the same event, the number of resulting chains is not influenced by $\overline{\Lambda}(s)$.

Subsequently, the previous measurement results are used to determine the impact of the OperationalScope complexity $\Lambda(OS)$ on the processing times of the chain derivation process. Due to the assumed simplifications in the context of the experiments, Equation 5-1 is simplified and the OperationalScope complexity is formulated as

$$\Lambda(OS) = \sum_{i=0}^{n_s} \Lambda(s_i) = n_s \cdot \overline{\Lambda}(s)\,.$$

In order to obtain the respective values for $\Lambda(OS)$, the scenario counts are thus multiplied by the average scenario complexities. The resulting outcome is plotted in Figure 5-2.

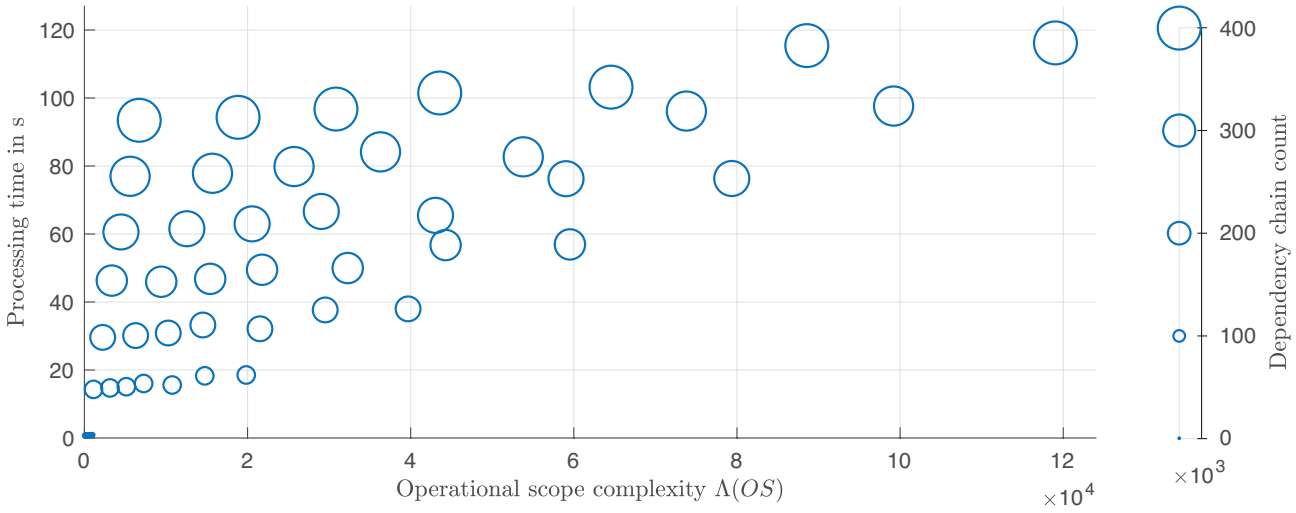


Figure 5-2: Measurement results in regard to a variation of the OperationalScope complexity $\Lambda(OS)$. Resulting processing times and dependency counts are plotted versus the OperationalScope complexities.

Once again, the individual measurement results are plotted as circles and the diameters of the circles provide information about the amount of resulting dependency chains. A certain pattern becomes apparent, but no clear dependence between the quantified OperationalScope complexity $\Lambda(OS)$ and the processing time is to be deduced. The pattern shows homogeneous progressions for certain measurement value clusters. On closer inspection, it becomes clear that these clusters originate from measurements with either consistent complexity $\overline{\Lambda}(s)$ or consistent scenario count n_s. It is therefore assumed that this pattern results from the significantly different impact of the sub-parameters. As shown in Figure 5-1, the scenario count has a much higher influence on the processing time than the average scenario complexity.

Next, the impact of the RuleCatalog complexity $\Lambda(RC)$ on the chain derivation process is investigated. This part of the evaluation is similar to the previous investigation of the OperationalScope complexity. Here, again, two independent setting parameters serve to vary the superordinate complexity $\Lambda(RC)$. These two parameters are the number of rules n_r and their complexity $\Lambda(r)$. Due to the met assumptions, in this case average Rule complexities $\overline{\Lambda}(r)$ are considered, too. Each RuleCatalog is thus modeled as a collection of a specific amount of identical, considerably average rules. The seven Rule variations listed in Table 5-2 are combined in seven different sets with Rule counts from $n_\mathrm{r} = 1$ to $n_\mathrm{r} = 60$. Again more increments or higher values are not realizable due to limited computing resources in the context of this work.

As before, the implemented software framework is charged with the resulting digital artifacts. In this case the same OperationalScope and the same SystemArchitecture are passed for all measurements. Again, each measurements is carried out ten times in order to counteract measurement inaccuracies. In Figure 5-3 the respective results are presented in the previously explained manner. Three diagrams are shown which visualize different views of the same measurement results. According to the measurements the average complexities $\overline{\Lambda}(r)$ and counts n_r of the included rules are plotted. Here, too, the diameters of the measured values depicted as circles represent the quantity of derived dependency chains. The measured values connected by lines in the two lower diagrams again have identical complexity values (left) and identical count values (right).

In this case, there are distinct dependencies between the two varied setting parameters and the processing time of the chain derivation process. In the lower left diagram it becomes clear that the processing time increases as soon as the Rule count is increased. Thereby, the depicted lines show different gradients. In addition, all measurements with identical Rule count also resulted in identical quantities of derived chains. Both is explained by remembering the structure of the chain derivation process. All rules of the RuleCatalog are run through iteratively and applied to a corresponding scenario. The rules have no influence on each other, hence the linear behavior. Additionally, due to the assumed simplifications, one and the same event in the scenarios is extracted independently of the *Rule* complexity $\overline{\Lambda}(r)$. Since the progressions of the different measurement series (connected by lines) differ, $\overline{\Lambda}(r)$ seems to have an influence on the processing time. This becomes even clearer in the lower right diagram. Here it is to be seen that with increasing *Rule* complexity $\overline{\Lambda}(r)$ the processing time increases as well. However, the amount of resulting dependency chains remains identical for the aforementioned reason. The gradient of the depicted lines seems to increase as the number of lines increases. Thus, it is not clear whether there is a linear relationship between the Rule complexity and processing time. For a justified statement, measurements with higher Rule counts would be necessary.

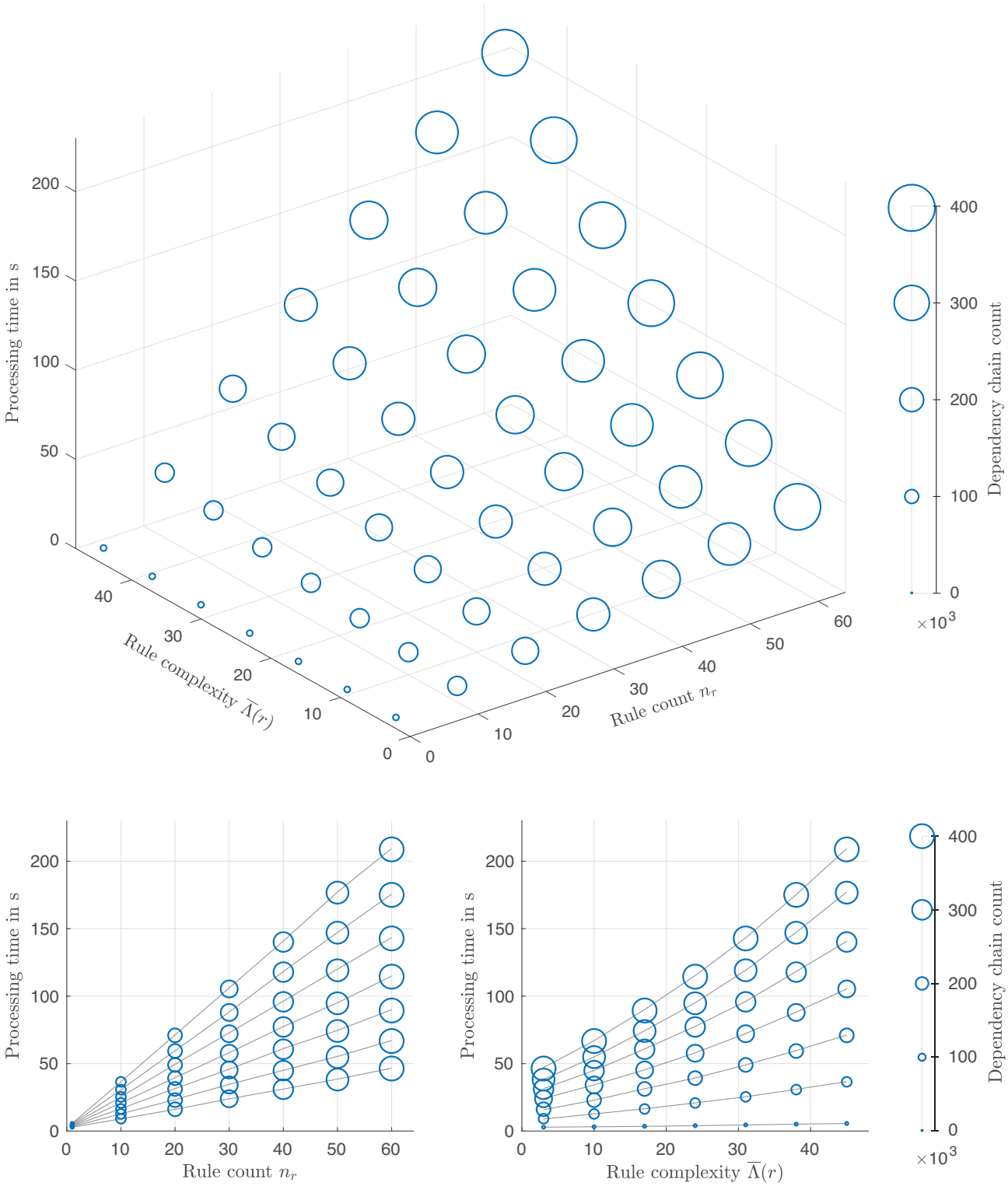


Figure 5-3: Measurement results based on varying Rule complexities $\Lambda(r)$ and Rule counts n_r. By instrumentalization of the implemented software framework measurements of the respective processing times and the count of derived dependency chains are conducted. The three diagrams depict the same measurements from three different perspectives.

The presented measurement results, which refer to the separate variation of the two setting parameters, are used again to draw further conclusions. In this case the impact of the RuleCatalog complexity $\Lambda(RC)$ on the processing time is investigated. Again, the met assumptions lead to a simplified measurement of the superordinate complexity metric, in this case $\Lambda(RC)$. Equation 5-2 is therefore simplified to

$$\Lambda(RC) = \sum_{i=0}^{n_\mathrm{r}} \Lambda(r_i) = n_\mathrm{r} \cdot \overline{\Lambda}(r)\,.$$

To derive the respective values for $\Lambda(RC)$, the Rule counts are multiplied by the average Rule complexities. The arising results are shown in Figure 5-4.

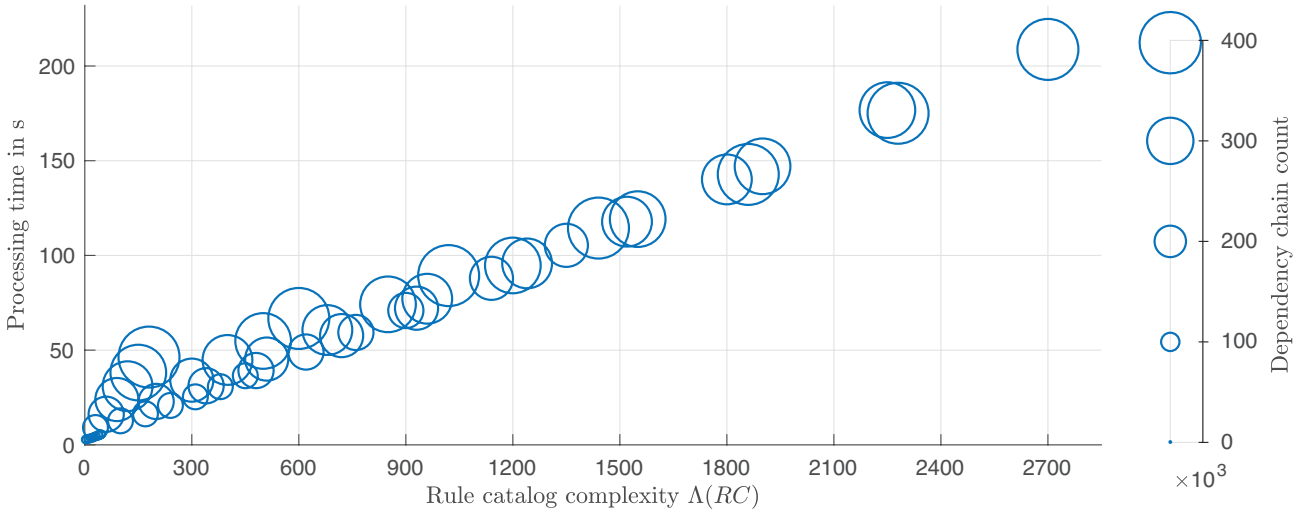


Figure 5-4: Measurement results in regard to a variation of the RuleCatalog complexity $\Lambda(RC)$. Resulting processing times and dependency counts are plotted versus the RuleCatalog complexities.

The way of plotting is identical to the diagram in Figure 5-2. This time the processing time is visualized over the RuleCatalog complexity $\Lambda(RC)$. The measured values are represented as circles with different diameters and the diameters refer to the number of resulting dependency chains. The result is comparable to the result in Figure 5-2. A similar, though more compressed pattern is visible. A distinct relation between the quantified RuleCatalog complexity $\Lambda(RC)$ and the processing time can only be assumed. The diagram again shows uniform progressions for certain measurement clusters. It becomes clear that these groups are derived from measurements with either consistent Rule Complexity $\overline{\Lambda}(r)$ or consistent Rule count n_r. This effect is again traced back to the different steep gradients in the previous graphs (Figure 5-3). The pattern is not as evident, but again the two sub-parameters have a different impact. However, the individual measured value clusters have approximately linear progressions.

As a third step, the impact of the SystemArchitecture $\Lambda(SA)$ on the chain derivation process is investigated. This turns out to be somewhat simpler than the two previous evaluation steps, since there is no splitting into two further setting parameters. The software framework is only charged with the seven *SystemArchitecture* variations from Table 5-3. Thereupon, the resulting processing times and dependency chain counts are plotted over the corresponding complexities. The measurement results are plotted in Figure 5-5.

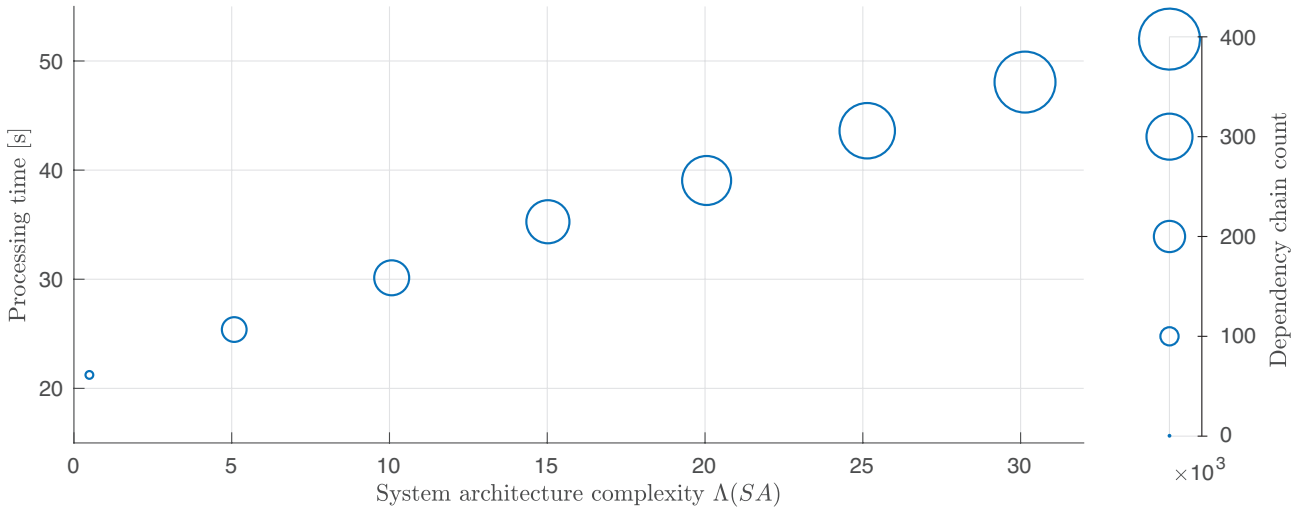


Figure 5-5: Measurement results in regard to a variation of the SystemArchitecture complexity $\Lambda(SA)$. Resulting processing times and dependency counts are plotted versus the SystemArchitecture complexities.

For the measurement representation the same notation as before is used. The measured values are represented by circles whose diameter visualizes the amount of derived dependency chains. It becomes clear that both processing time and chain count depend on the SystemArchitecture complexity $\Lambda(SA)$ in the considered value range. As complexity increases, both result indicators increase as well. Here, the increase of the processing time is linear. This is attributed to the way in which $\Lambda(SA)$ is quantified. The measurement of the complexity of a SystemArchitecture artifact in the context of this paper is based on the average of all possible paths between the main skill nodes and the sensor resp. actor nodes. Since each found path is handled individually in the chain derivation process, the processing time increases accordingly. In addition, each of these paths results in a corresponding dependency chain. Hence the distinct dependency between $\Lambda(SA)$ and the dependency chain count arises.

What conclusions may be drawn from the experiments regarding the load scalability of the approach? This question does not seem to be fully answerable on the basis of the conducted experiments. All quantified setting parameters seem to have an influence on the processing times. If one of the parameters n_s, $\overline{\Lambda}(s)$, n_r, $\overline{\Lambda}(r)$ or $\Lambda(SA)$ is varied individually in the considered value range, approximately linear correlations are to be recognized. However, it also becomes clear that the individual setting parameters are mutually dependent. The work load magnitude of the software framework is influenced by all parameters in different ways. This makes it difficult to estimate the overall load scalability. The developed quantification metrics $\Lambda(r)$ and $\Lambda(SA)$ are estimated to be representative, since the corre-

sponding measurement processes provide comprehensible results. However, the measurement of the scenario complexity $\Lambda(s)$ is questioned. Here, the results of the measurements lead to progressions that are only to be explained by conjecture. In summary, the experiments provided information on the extent to which the individual setting parameters influence the processing time and the number of resulting dependency chains. Ultimately, however, the conclusion is drawn that further, more detailed investigations must be carried out in order to derive a well-founded statement on the load scalability of the entire approach.

6 Conclusion and Outlook

The approach presented in this work allows the automatic derivation of dependency chains in systems for autonomous driving. Hereby, dependencies between ontology based scenario representations and components of a graph-based modeled system architecture are determined. This aims at supporting design decisions in the development process by a more detailed view on overall systems. In addition, this intends to improve the traceability of decisions. The main focus of this work has been the development of a suitable solution concept. On the basis of the developed concept, an implementation has been performed in order to provide a proof of concept.

Initially, a meta model, which describes the addressed correlations based on established terminologies and definitions, has been proposed. Thereby, it has been elaborated that particular skill descriptions in a system serve as connecting elements between scenario representations and system components in the context of this work. From the meta model, three key challenges have been extracted regarding the intended solution. These have been addressed by individual sub-concepts resp. partial solutions.

On the one hand, an ontology has been developed in order to enable the creation of the required scenario presentations. The proposed ontology facilitates the modeling of dedicated driving scenarios and focuses on the representation of relation concepts, which serve to semantically derive particular requirements. On the other hand, a special representation method of system architectures, a so-called task-chain-pattern skill graph representation, has been suggested. This modeling alternative enables the representation of links to the skills of a system as well as the extraction of automatic paths to system components within an architecture. Finally, the concept of a chain derivation engine has been developed and presented in this work. The chain derivation engine is considered the algorithmic core of the approach and is based on semantic rules as well as arithmetic functions, both of which facilitate the final derivation of the dependency chains.

Subsequently, implemented examples for all parts of the approach have been presented. These have then been included in a common software framework and set in relation to each other.

Thereby, it has been clarified at which steps of the modeling process expert knowledge needs to be incorporated. In addition, necessary alignments between the involved digital artifacts have been examined. A final holistic example has been utilized to demonstrate the functionality of the chain derivation engine and to outline elaborated constraints. Here, both the inherent program routines and the implemented analysis functionalities have been examined. The derived dependency chains enable to express requirements for particular system components. They additionally facilitate the reveal of conflicts between a vehicle's intended operational scope and predefined system components.

Finally, an evaluation has been conducted. Therein, the implemented software framework has been instrumentalized to examine the load scalability of the approach. Fictitious scenario catalogs, system architectures and semantic rules with different complexities have been generated. The software framework has then been charged with these data sources. Following, measured processing times as well as quantities of derived dependency chains have been compared. Based in this, correlations be-

tween several setting parameters, e.g. the amount of semantic rules or the complexity of the system's architecture, have been identified.

The developed approach is understood as a prototype and does not claim to be universally applicable. Examples have been used to demonstrate what needs to be considered for an application. However, in order to test the applicability beyond the examples, further investigations are necessary. Due to identified dependencies between the ontology, the semantic rules and the modeled skill graph system architecture, it is assumed that an application of the approach by multiple team members is challenging. In order to investigate this, it is recommended to include multiple persons in the modeling of the individual sub-structures and to thereby examine the applicability.

It became apparent that expert knowledge must be incorporated into the sub-steps of the approach in order to facilitate the automatic derivation of dependencies. Therefore, in addition to the elaborated examples, the modeling effort in regard to a real system should be evaluated. It is assumed that the experimental use of the approach in a practical project is necessary to address this issue.

Besides, it is to be investigated whether the overall modeling effort could be reduced. Here, it should be examined whether the partial steps of the suggested approach are improvable by further process concepts. For example, an enhanced utilization of the semantic coherences formulated by the ontology could be beneficial.

Moreover, when modeling the system's architecture, further information should be included in order to enable the implementation of additional analysis functionalities. For instance, it should be assessed whether the nodes or edges in the graph could be assigned with latency times of the particular sub-systems. If these could be included in the derivation of dependency chains, more enhanced statements about system requirements may be conceivable. For the same reason, redundancies within the system's architecture should be modeled concretely and included in the chain derivation process.

It is also assumed that the inference capabilities of the ontology offer further potential in regard to the suggested approach. Here, especially the axioms formulated in SWRL should receive further attention. Some of the language's capabilities were exploited in the context of this work, however, it is presumed that further utilization concepts are plausible.

The feasibility of the developed approach has been tested in relation to a limited amount of modeled examples. One of these examples has been presented in the course of this work. It is suggested to examine to which extent further, deviating examples may be addressed by the approach.

A Appendix

A.1 Software Framework Implementation

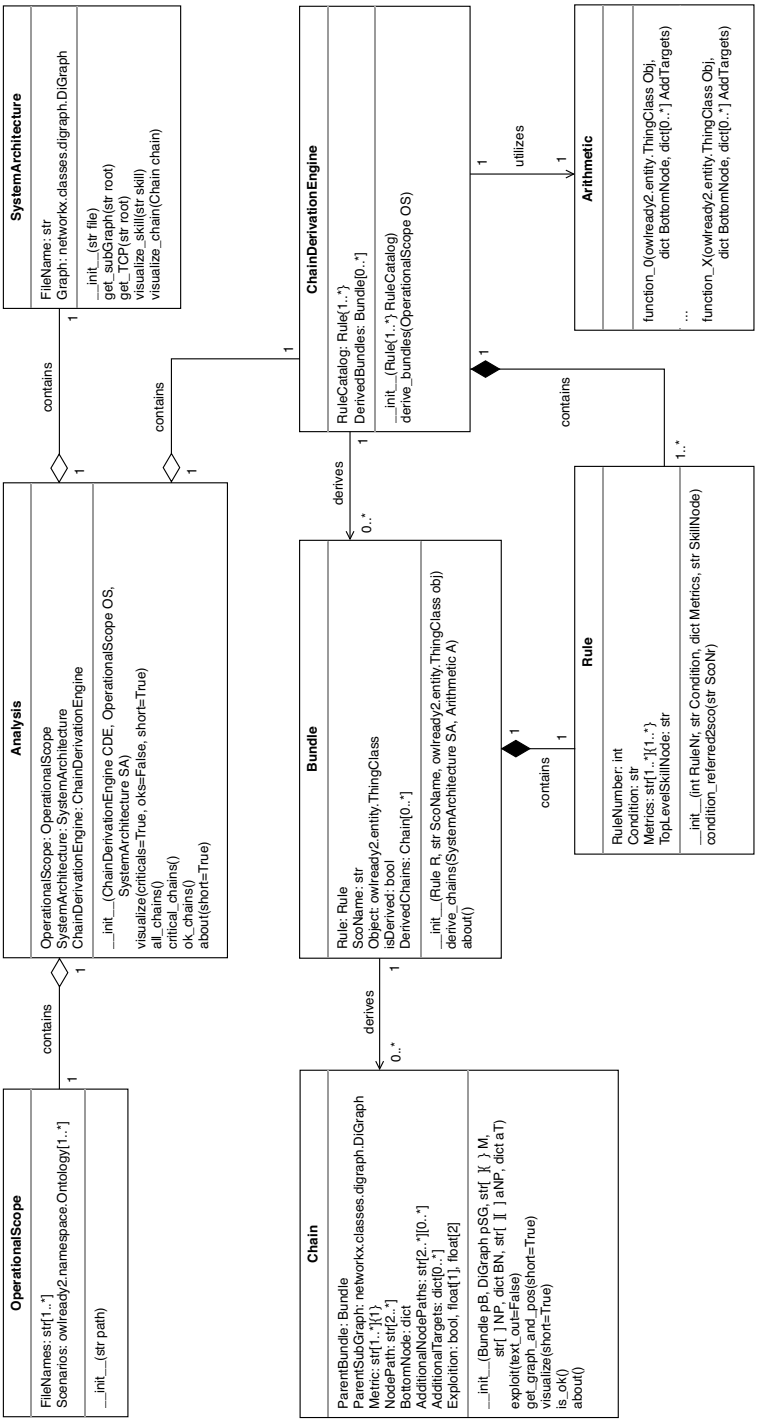


Figure A-1: UML class diagram of the implemented software framework. The classes OperationalScope, SystemArchitecture and ChainDerivationEngine represent the three key parts of the elaborated concept and are part of the Analysis class. The ChainDerivationEngine class contains Rules and utilizes a respective Arithmetic. It derives instances of the Bundle class from which subsequently instances of the Chain class are derived. Each Bundle contains exactly one Rule and serves to structure the derivation process.

A.2 Developed Ontology

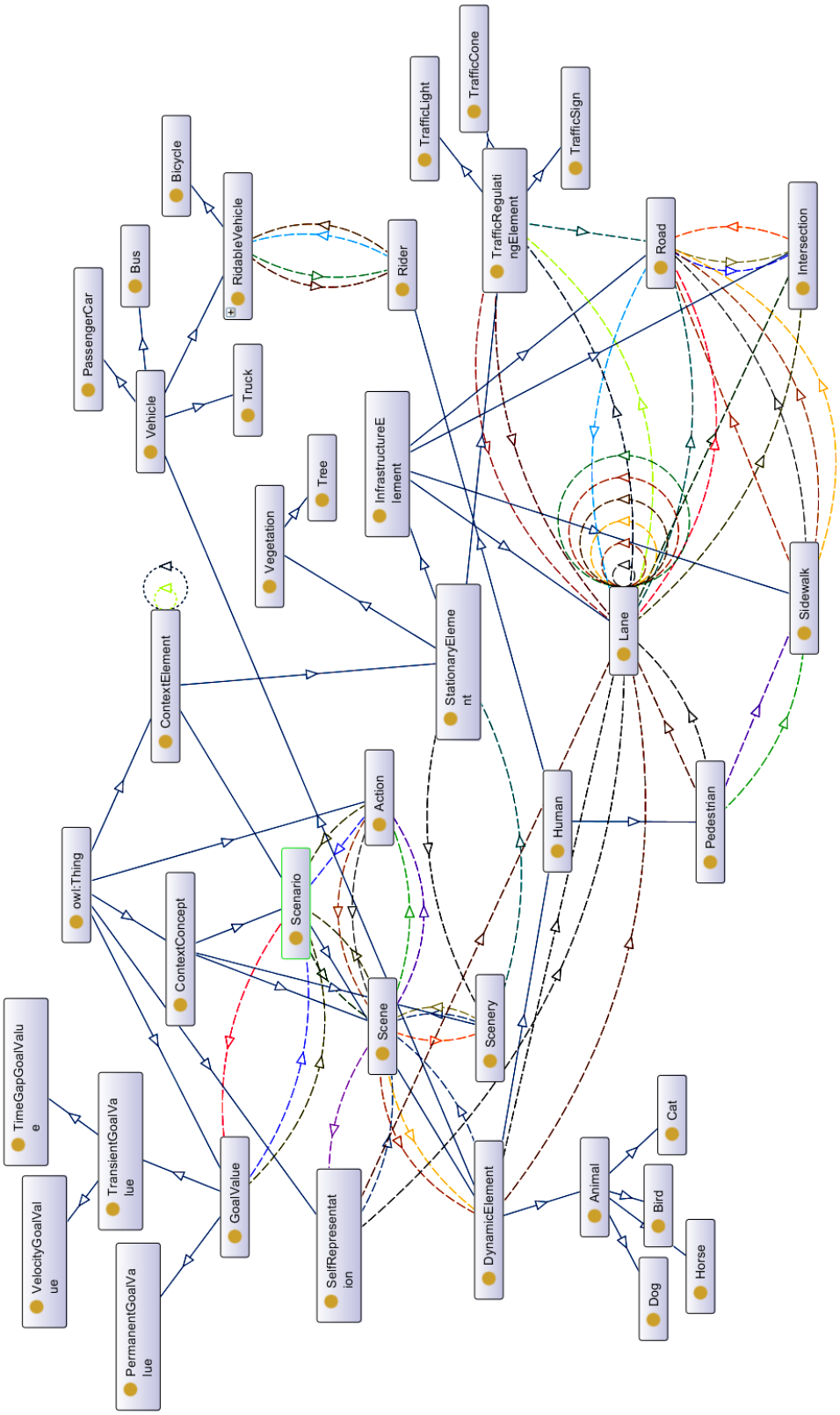
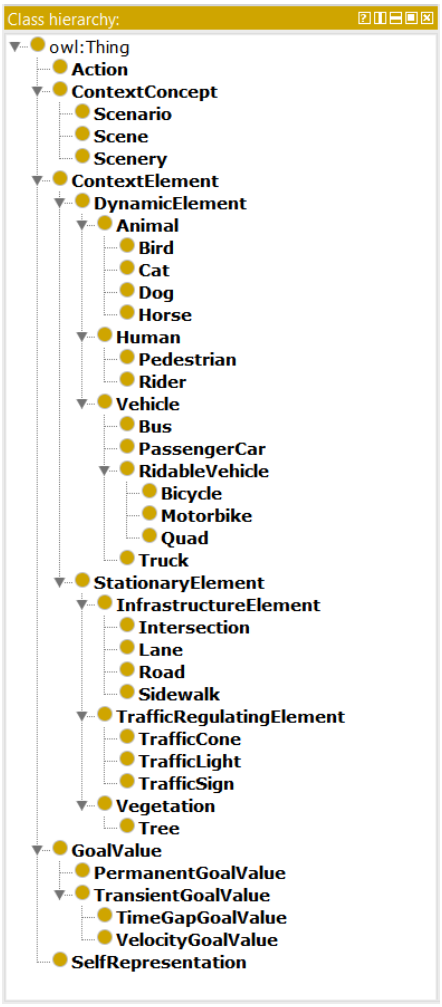
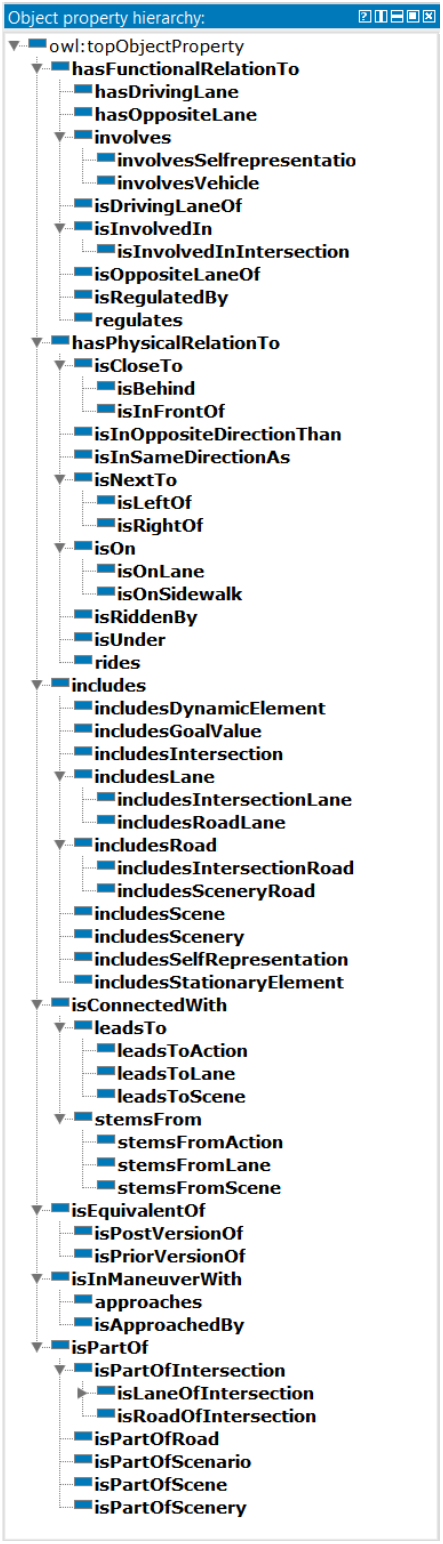


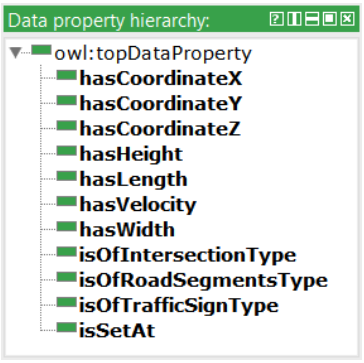
Figure A-2: Overall structure of the developed ontology depicted as a graph via the *OntoGraf* plugin in *Protégé*.



(a) Class hierarchy



(b) Relation hierarchy



(c) Property list

Figure A-3: Complete listing of the specified classes, relations and properties of the developed ontology as excerpts from *Protégé*. The ontology's expressiveness is implied, its comprehensive structure, however, is only explicable by an extensive disclosure of further inherited specifications.

A.3 Developed Ontology Axioms – SWRL Rules

Road(?R1) ^ Road(?R2) ^ Lane(?L1) ^ Lane(?L2) ^ includes(?R1, ?L1) ^ isPriorVersionOf(?R1, ?R2) ^ isPriorVersionOf(?L1, ?L2) -> includesRoadLane(?R2, ?L2)

Scene(?SCN1) ^ Scene(?SCN2) ^ Scenery(?SCY1) ^ Scenery(?SCY2) ^ includes(?SCN1, ?SCY1) ^ isPriorVersionOf(?SCN1, ?SCN2) ^ isPriorVersionOf(?SCY1, ?SCY2) -> includesScenery(?SCN2, ?SCY2)

Scene(?SCN1) ^ Scene(?SCN2) ^ SelfRepresentation(?SR1) ^ SelfRepresentation(?SR2) ^ includes(?SCN1, ?SR1) ^ isPriorVersionOf(?SCN1, ?SCN2) ^ isPriorVersionOf(?SR1, ?SR2) -> includesSelfRepresentation(?SCN2, ?SR2)

Scenery(?SCY1) ^ Scenery(?SCY2) ^ ContextElement(?E1) ^ ContextElement(?E2) ^ includes(?SCY1, ?E1) ^ isPriorVersionOf(?SCY1, ?SCY2) ^ isPriorVersionOf(?E1, ?E2) -> includes(?SCY2, ?E2)

SelfRepresentation(?SR) ^ Lane(?L1) ^ Lane(?L2) ^ Road(?R) ^ isDrivingLaneOf(?L1, ?SR) ^ isPartOf(?L1, ?R) ^ isPartOf(?L2, ?R) ^ isInSameDirectionAs(?L1, ?L2) -> isDrivingLaneOf(?L2, ?SR)

Vehicle(?V) ^ Lane(?L1) ^ Lane(?L2) ^ Road(?R) ^ isDrivingLaneOf(?L1, ?V) ^ isPartOf(?L1, ?R) ^ isPartOf(?L2, ?R) ^ isInSameDirectionAs(?L1, ?L2) -> isDrivingLaneOf(?L2, ?V)

SelfRepresentation(?SR) ^ Lane(?L) ^ isOn(?SR, ?L) -> isDrivingLaneOf(?L, ?SR)

Vehicle(?V) ^ Lane(?L) ^ isOn(?V, ?L) -> isDrivingLaneOf(?L, ?V)

Vehicle(?V1) ^ Vehicle(?V2) ^ Road(?R) ^ isOn(?V1, ?R) ^ isOn(?V2, ?R) ^ differentFrom(?V1, ?V2) -> isInManeuverWith(?V1, ?V2)

SelfRepresentation(?SR) ^ Vehicle(?V) ^ Road(?R) ^ isOn(?SR, ?R) ^ isOn(?V, ?R) -> isInManeuverWith(?SR, ?V)

Lane(?L1) ^ Lane(?L2) ^ Road(?R) ^ Intersection(?I) ^ isPartOf(?L1, ?R) ^ isPartOf(?L2, ?R) ^ isInSameDirectionAs(?L1, ?L2) ^ isIncomingLaneOfIntersection(?L1, ?I) -> isIncomingLaneOfIntersection(?L2, ?I)

Lane(?L1) ^ Lane(?L2) ^ Road(?R) ^ Intersection(?I) ^ isPartOf(?L1, ?R) ^ isPartOf(?L2, ?R) ^ isInOppositeDirectionThan(?L1, ?L2) ^ isOutgoingLaneOfIntersection(?L1, ?I) -> isIncomingLaneOfIntersection(?L2, ?I)

SelfRepresentation(?SR) ^ Lane(?L) ^ Intersection(?I) ^ isOn(?SR, ?L) ^ isIncomingLaneOfIntersection(?L, ?I) -> isInvolvedInIntersection(?SR, ?I)

Vehicle(?V) ^ Lane(?L) ^ Intersection(?I) ^ isOn(?V, ?L) ^ isIncomingLaneOfIntersection(?L, ?I) -> isInvolvedInIntersection(?V, ?I)

Vehicle(?V1) ^ Vehicle(?V2) ^ Lane(?L1) ^ Lane(?L2) ^ isOn(?V1, ?L1) ^ isOn(?V2, ?L2) ^ isLeftOf(?L1, ?L2) -> isLeftOf(?V1, ?V2)

SelfRepresentation(?SR) ^ Vehicle(?V2) ^ Lane(?L1) ^ Lane(?L2) ^ isOn(?SR, ?L1) ^ isOn(?V2, ?L2) ^ isLeftOf(?L1, ?L2) -> isLeftOf(?SR, ?V2)

Vehicle(?V) ^ Lane(?L) ^ Road(?R) ^ isOn(?V, ?L) ^ isPartOf(?L, ?R) -> isOn(?V, ?R)

SelfRepresentation(?SR) ^ Lane(?L) ^ Road(?R) ^ isOn(?SR, ?L) ^ isPartOf(?L, ?R) -> isOn(?SR, ?R)

SelfRepresentation(?SR) ^ Lane(?L1) ^ Lane(?L2) ^ Road(?R) ^ isDrivingLaneOf(?L1, ?SR) ^ isPartOf(?L1, ?R) ^ isPartOf(?L2, ?R) ^ isInOppositeDirectionThan(?L1, ?L2) -> isOppositeLaneOf(?L2, ?SR)

Vehicle(?V) ^ Lane(?L1) ^ Lane(?L2) ^ Road(?R) ^ isDrivingLaneOf(?L1, ?V) ^ isPartOf(?L1, ?R) ^ isPartOf(?L2, ?R) ^ isInOppositeDirectionThan(?L1, ?L2) -> isOppositeLaneOf(?L2, ?V)

Lane(?L1) ^ Lane(?L2) ^ Road(?R) ^ Intersection(?I) ^ isPartOf(?L1, ?R) ^ isPartOf(?L2, ?R) ^ isInSameDirectionAs(?L1, ?L2) ^ isOutgoingLaneOfIntersection(?L1, ?I) -> isOutgoingLaneOfIntersection(?L2, ?I)

Lane(?L1) ^ Lane(?L2) ^ Road(?R) ^ Intersection(?I) ^ isPartOf(?L1, ?R) ^ isPartOf(?L2, ?R) ^ isInOppositeDirectionThan(?L1, ?L2) ^ isIncomingLaneOfIntersection(?L1, ?I) -> isOutgoingLaneOfIntersection(?L2, ?I)

Lane(?L1) ^ Lane(?L2) ^ Road(?R) ^ isPartOf(?L1, ?R) ^ hasPhysicalRelationTo(?L1, ?L2) -> isPartOfRoad(?L2, ?R)

Scene(?SCN1) ^ Scene(?SCN2) ^ Scenario(?SCO) ^ isPartOf(?SCN1, ?SCO) ^ isEquivalentOf(?SCN1, ?SCN2) -> isPartOfScenario(?SCN2, ?SCO) ^ includesScene(?SCO, ?SCN2)

ContextElement(?cE1) ^ ContextElement(?cE2) ^ Scenery(?S) ^ hasPhysicalRelationTo(?cE1, ?cE2) ^ isPartOf(?cE1, ?S) -> isPartOfScenery(?cE2, ?S)

SelfRepresentation(?SR) ^ ContextElement(?cE) ^ Scenery(?S) ^ hasPhysicalRelationTo(?SR, ?cE) ^ isPartOf(?SR, ?S) -> isPartOfScenery(?cE, ?S)

SelfRepresentation(?SR) ^ ContextElement(?cE) ^ Scenery(?S) ^ hasPhysicalRelationTo(?cE, ?SR) ^ isPartOf(?cE, ?S) -> isPartOfScenery(?SR, ?S)

Scene(?SCN1) ^ Action(?A) ^ Scene(?SCN2) ^ leadsTo(?SCN1, ?A) ^ leadsTo(?A, ?SCN2) -> isPriorVersionOf(?SCN1, ?SCN2)

Scene(?SCN1) ^ Scene(?SCN2) ^ Scenery(?SCY1) ^ Scenery(?SCY2) ^ includesScenery(?SCN1, ?SCY1) ^ includesScenery(?SCN2, ?SCY2) ^ isPriorVersionOf(?SCN1, ?SCN2) -> isPriorVersionOf(?SCY1, ?SCY2)

Vehicle(?V1) ^ Vehicle(?V2) ^ Lane(?L1) ^ Lane(?L2) ^ isOn(?V1, ?L1) ^ isOn(?V2, ?L2) ^ isRightOf(?L1, ?L2) -> isRightOf(?V1, ?V2)

SelfRepresentation(?SR) ^ Vehicle(?V2) ^ Lane(?L1) ^ Lane(?L2) ^ isOn(?SR, ?L1) ^ isOn(?V2, ?L2) ^ isRightOf(?L1, ?L2) -> isRightOf(?SR, ?V2)

Lane(?L) ^ Road(?R) ^ Intersection(?I) ^ isPartOf(?L, ?R) ^ isPartOf(?L, ?I) -> isRoadOfIntersection(?R, ?I)

Lane(?L1) ^ Lane(?L2) ^ Road(?R1) ^ Road(?R2) ^ isPartOf(?L1, ?R1) ^ isPartOf(?L2, ?R2) ^ isIncomingLaneOfIntersection(?L1, ?I) ^ isOutgoingLaneOfIntersection(?L2, ?I) ^ differentFrom(?R1, ?R2) -> leadsToLane(?L1, ?L2)

Figure A-4: List of all specified axioms resp. SWRL rules in the context of the developed ontology. The rules facilitate the representation of complex relations in the ontology and are utilized in the inferencing process. They are particularly modeled to enable an alignment of the expressiveness of eventual knowledge bases.

A.4 Automatically Derived Dependency Chains

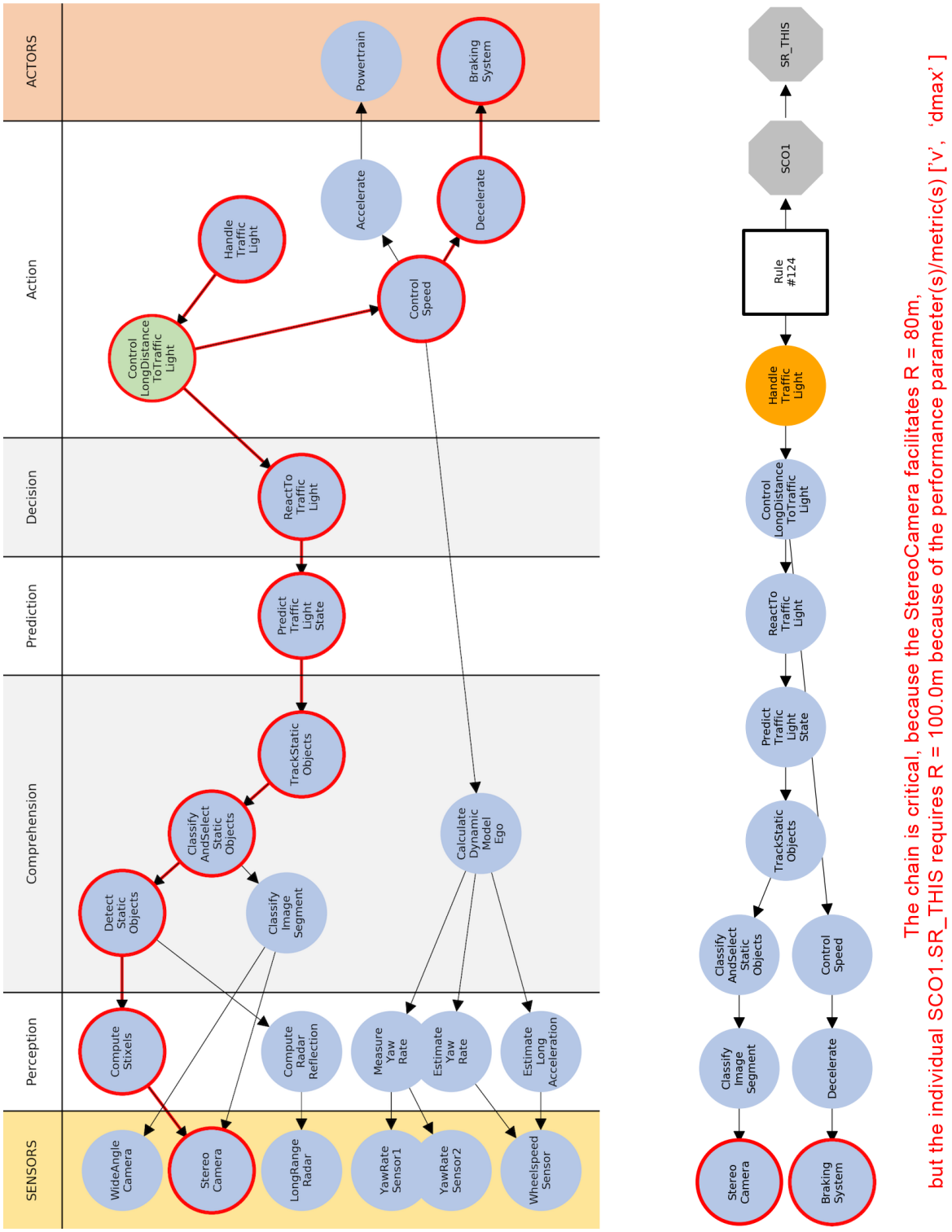


Figure A-5: One of four automatically derived dependency chains in form of the output of a developed analysis functionality. Both the chain in regard to the respective task-chain-pattern skill graph as well as the complete dependency chain reaching from a particular object to two system components are depicted. Additionally, an automatically derived statement in natural language is shown. In this case, a conflict is recognized, thus, the chain is classified as critical.

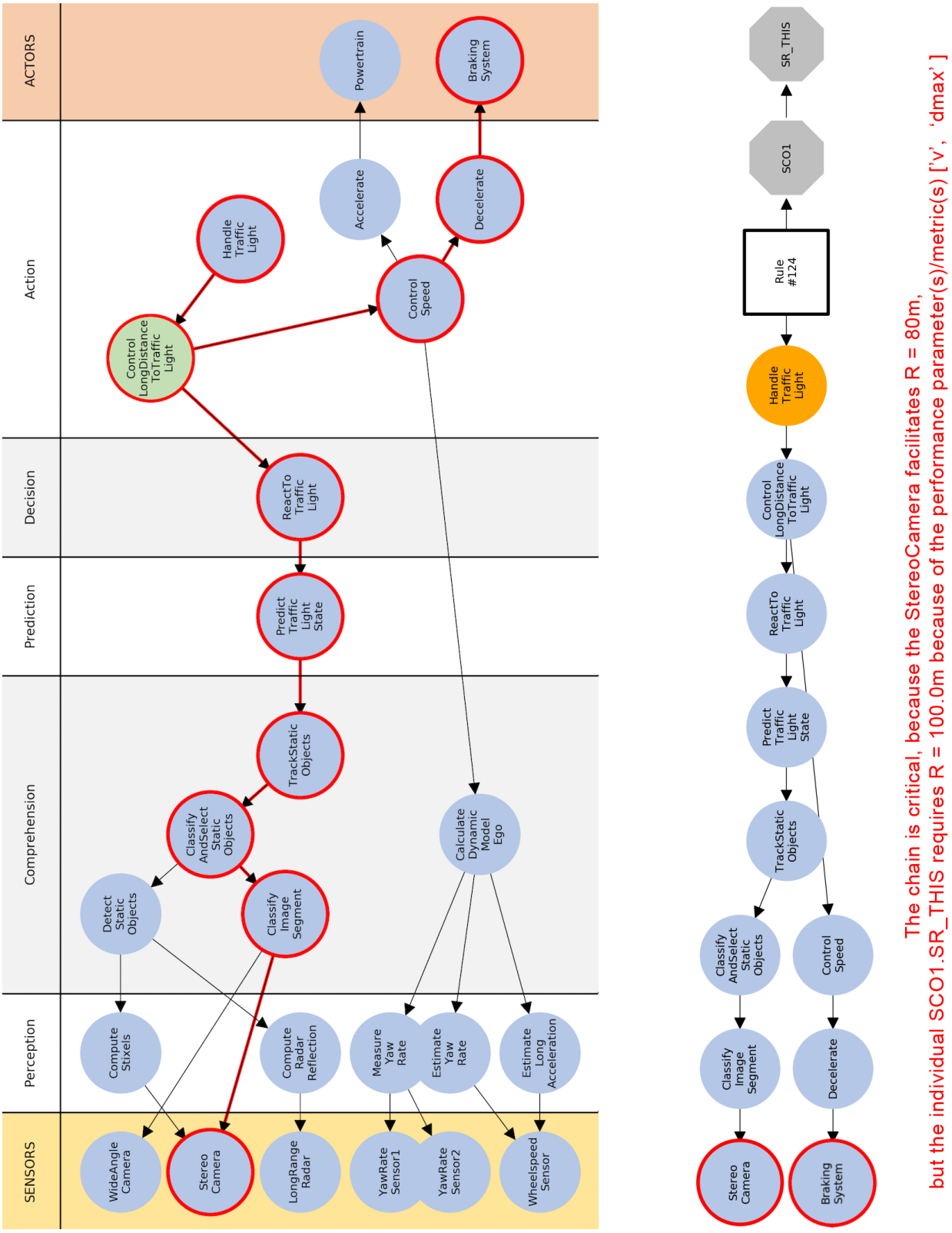


Figure A-6: One of four automatically derived dependency chains in form of the output of a developed analysis functionality. Both the chain in regard to the respective task-chain-pattern skill graph as well as the complete dependency chain reaching from a particular object to two system components are depicted. Additionally, an automatically derived statement in natural language is shown. In this case, a conflict is recognized, thus, the chain is classified as critical.

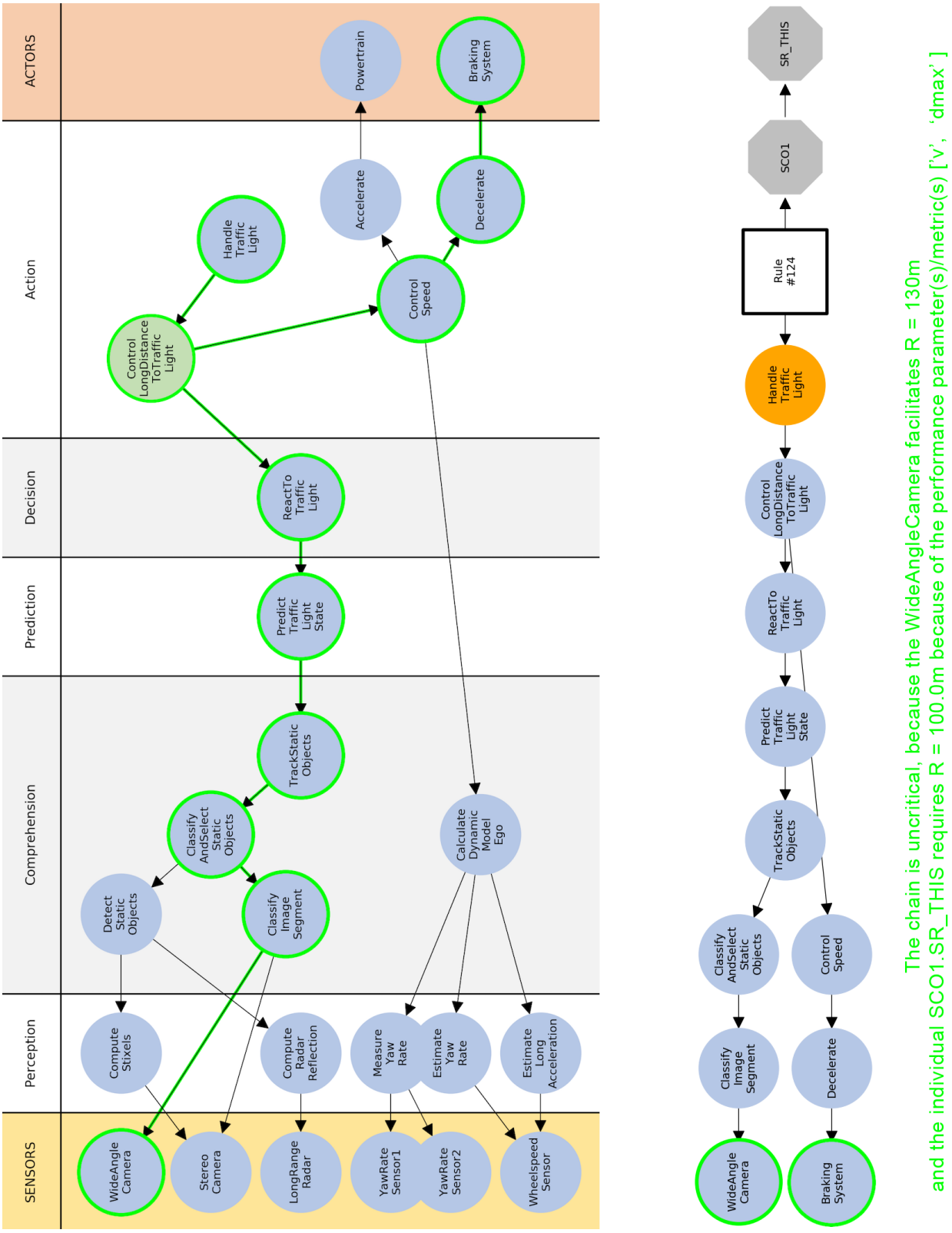


Figure A-7: One of four automatically derived dependency chains in form of the output of a developed analysis functionality. Both the chain in regard to the respective task-chain-pattern skill graph as well as the complete dependency chain reaching from a particular object to two system components are depicted. Additionally, an automatically derived statement in natural language is shown. In this case, no conflict is recognized, thus, the chain is classified as uncritical.

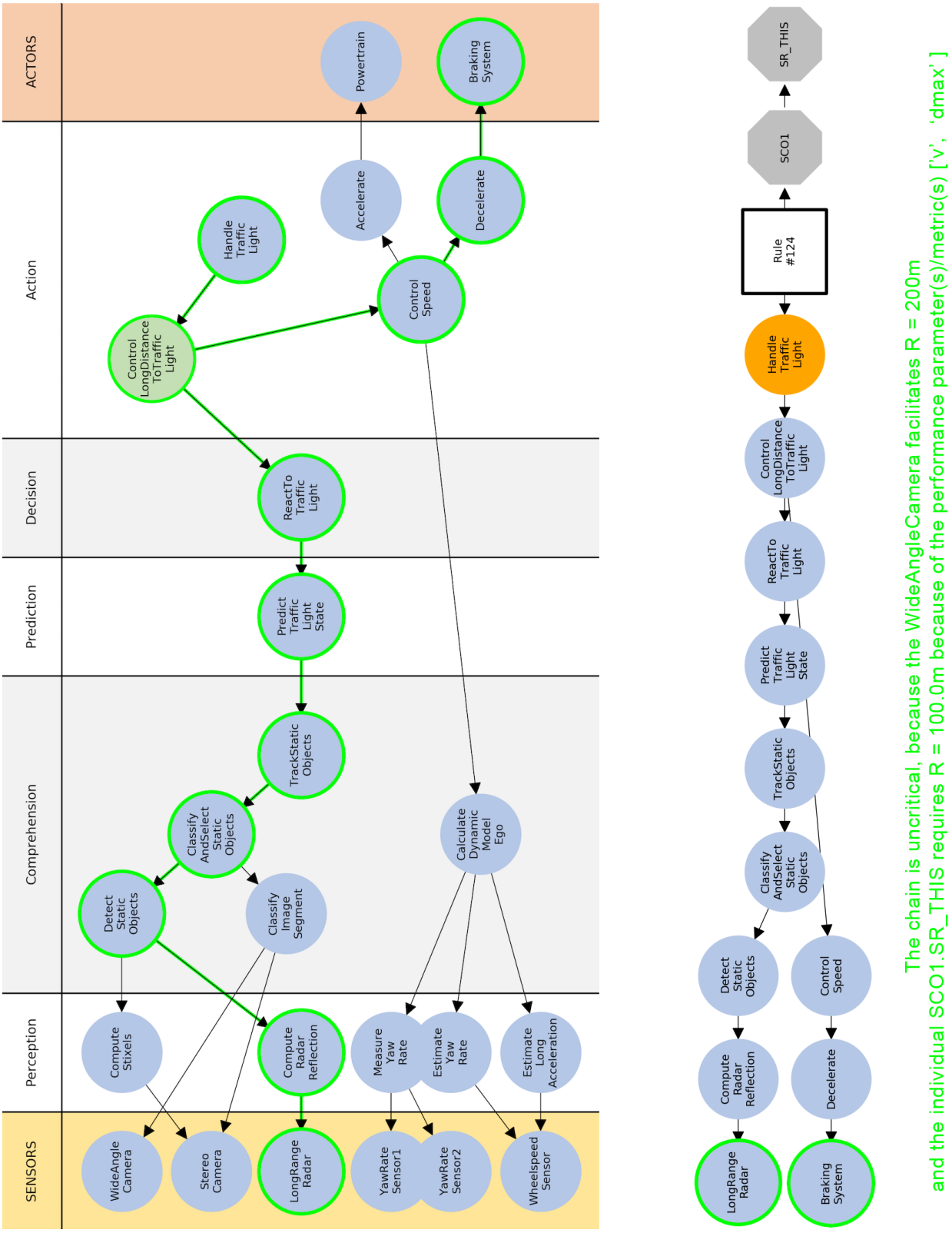


Figure A-8: One of four automatically derived dependency chains in form of the output of a developed analysis functionality. Both the chain in regard to the respective task-chain-pattern skill graph as well as the complete dependency chain reaching from a particular object to two system components are depicted. Additionally, an automatically derived statement in natural language is shown. In this case, no conflict is recognized, thus, the chain is classified as uncritical.

Bibliography

Amersbach, C.; Winner, H.: Functional Decomposition (2017)
Amersbach, C.; Winner, H.: Functional Decomposition: An Approach to Reduce the Approval Effort for Highly Automated Driving, in: 8. Tagung Fahrerassistenz, München, 2017

Bagschik, G. et al.: Ontology based Scene Creation for Automated Vehicles (2018)
Bagschik, G.; Menzel, T.; Maurer, M.: Ontology based Scene Creation for the Development of Automated Vehicles, in: IEEE Intelligent Vehicles Symposium, pp. 1813–1820, 2018

Bagschik, G. et al.: Architecture Framework for Automated Vehicles (2018)
Bagschik, G.; Nolte, M.; Ernst, S.; Maurer, M.: A System's Perspective Towards an Architecture Framework for Safe Automated Vehicles, in: IEEE International Conference on Intelligent Transportation Systems, 2018

Bermejo-Alonso, J. et al.: Ontology Engineering for Autonomous Systems (2013)
Bermejo-Alonso, J.; Sanz, R.; Rodríguez, M.; Hernández, C.: Ontology Engineering for the Autonomous Systems Domain, in: Communications in Computer and Information Science, vol. 348, pp. 263–277, 2013

Bermejo, J.: A Simplified Guide to Create an Ontology (2007)
Bermejo, J.: A Simplified Guide to Create an Ontology, in: The Autonomous Systems Laboratory, Universidad Politécnica de Madrid, Madrid, 2007

Bondi, A. B.: Characteristics of Scalability and Their Impact on Performance (2000)
Bondi, A. B.: Characteristics of Scalability and Their Impact on Performance, in: Proceedings of the 2nd International Workshop on Software and Performance, pp. 195–203, 2000

Borst, W. N.: Diss., Construction of Engineering Ontologies (1997)
Borst, W. N.: Construction of Engineering Ontologies for Knowledge Sharing and Reuse, Dissertation University of Twente, Centre for Telematics and Information Technology, Enschede, 1997

Busse, J. et al.: Actually, What Does "Ontology" Mean? (2015)
Busse, J.; Humm, B. G.; Lübbert, C.; Moelter, F.; Reibold, A.; Rewald, M.; Schlüter, V.; Seiler, B.; Tegtmeier, E.; Zeh, T.: Actually, What Does Ontology Mean? – A Term Coined by Philosophy in the Light of Different Scientific Disciplines, in: Journal of Computing and Information Technology, vol. 23, no. 1, pp. 29–41, 2015

Castro, C.: Human Factors in Driving (2008)
Castro, C.: Human Factors of Visual and Cognitive Performance in Driving, CRC Press, Taylor und Francis Group, LLC, Boca Raton, 2008

Dajsuren, Y. et al.: Correspondence Rules for Architecture Views (2014)
Dajsuren, Y.; Gerpheide, C. M.; Serebrenik, A.; Wijs, A.; Vasilescu, B.; van den Brand, M. G. J.: Formalizing Correspondence Rules for Automotive Architecture Views, in: International ACM Sigsoft Conference on Quality of Software Architectures, 2014

El-Rewini, H.; Abd-El-Barr, M.: Advanced Computer Architecture Programming (2004)
El-Rewini, H.; Abd-El-Barr, M.: Advanced Computer Architecture and Parallel Programming, John Wiley & Sons, Hoboken, 2004

Endsley, M. R.: Situation Awareness in Dynamic Systems (1995)
Endsley, M. R.: Towards a Theory of Situation Awareness in Dynamic Systems, in: Human Factors, vol. 37, pp. 32–64, 1995

Falconer, S.: OntoGraf (2013)
Falconer, S.: OntoGraf, URL: https://protegewiki.stanford.edu/wiki/OntoGraf, 2013, accessed 05.04.2019

Geyer, S. et al.: Ontology for Test and Use-case Catalogues (2014)
Geyer, S.; Baltzer, M.; Franz, B.; Hakuli, S.; Kauer, M.; Kienle, M.; Meier, S.; Weissgerber, T.; Bengler, K.; Bruder, R.; Flemisch, F.; Winner, H.: Concept and Development of a Unified Ontology for Generating Test and Use-case Catalogues for Assisted and Automated Vehicle Guidance, in: IET Intelligent Transport Systems, vol. 8, iss. 3, pp. 183–189, 2014

Gómez-Pérez, A. et al.: Ontological Engineering (2004)
Gómez-Pérez, A.; Fernández-López, M.; Corcho, O.: Ontological Engineering – with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web, Springer, London, 2004

Grimnes, G. A.: RDFLib (2018)
Grimnes, G. A.: RDFLib, URL: https://github.com/RDFLib/rdflib/, 2018, accessed 09.04.2019

Gruber, T. R.: A Translation Approach to Portable Ontology Specification (1993)
Gruber, T. R.: A Translation Approach to Portable Ontology Specification, in: Knowledge Acquisition, vol. 5, no. 2, pp. 199–220, 1993

Guarino, N. et al.: What is an Ontology? (2009)
Guarino, N.; Oberle, D.; Staab, S.: What is an Ontology?, in: Handbook on Ontologies, Springer, Berlin, pp. 1–17, 2009

Hebisch, E. et al.: Architecture Trace Diagrams (2015)
Hebisch, E.; Book, M.; Gruhn, V.: Scenario-based Architecting with Architecture Trace Diagrams, in: International Workshop on Twin Peaks of Requirements and Architecture, vol. 5, pp. 513–532, 2015

Hill, M. D.: What is Scalability? (1990)
Hill, M. D.: What is Scalability?, in: ACM SIGARCH Computer Architecture News, vol. 18, iss. 4, pp. 18–21, 1990

Hitzler, P. et al.: Foundations of Semantic Web Technologies (2010)
Hitzler, P.; Krötzsch, M.; Rudolph, S.: Foundations of Semantic Web Technologies, CRC Press, Taylor und Francis Group, LLC, Boca Raton, 2010

Hitzler, P. et al.: Semantic Web (2008)
Hitzler, P.; Krötzsch, M.; Rudolph, S.; Sure, Y.: Semantic Web – Grundlagen, Springer, Berlin, Heidelberg, 2008

Horrocks, I. et al.: SWRL: A Semantic Web Rule Language (2004)
Horrocks, I.; Patel-Schneider, P. F.; Boley, H.; Tabet, S.; Grosof, B.; Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML, W3C Member Submission, URL: https://www.w3.org/Submission/SWRL, 2004, accessed 09.03.2019

Kaiya, H. et al.: Using Domain Ontology for Requirements Elicitation (2006)
Kaiya, H.; Saeki, M.: Using Domain Ontology as Domain Knowledge for Requirements Elicitation, in: IEEE International Requirements Engineering Conference, vol. 14, 2006

Karbe, T. et al.: State of the Art for Automotive Ontology (2014)
Karbe, T.; Harting, K.; Leitner, A.; Melzi, A.; Ekelin, C.; Can, Ö.; Reuter, C.; du Pontavice, P.; Settelmeier, J.: State of the Art for Automotive Ontology, Critical System Engineering Acceleration, Deliverable no. D308.010, Technische Universität Berlin, Berlin, 2014

Lamy, J. B.: Owlready: Ontology-oriented Programming in Python (2017)
Lamy, J. B.: Owlready: Ontology-oriented Programming in Python with Automatic Classification and High Level Constructs for Biomedical Ontologies, in: Artificial Intelligence in Medicine, pp. 11–28, 2017

Lasheras, J. et al.: Security Requirements Based on an Ontology (2009)
Lasheras, J.; Valencia-García, R.; Fernandez-Breis, J. T.; Toval, A.: Modelling Reusable Security Requirements Based on an Ontology Framework, in: Journal of Research and Practice in Information Technology, vol. 41, no. 2, pp. 119–133, 2009

Maier, F. et al.: Ontology-Based Design Optimisation Support (2008)
Maier, F.; Mayer, W.; Stumptner, M.; Mühlenfeld, A.: Ontology-Based Process Modelling for Design Optimisation Support, in: Design Computing and Cognition, Springer, Dordrecht, pp. 513–532, 2008

Matthaei, R.; Maurer, M.: Autonomous Driving – a Top-Down-Approach (2015)
Matthaei, R.; Maurer, M.: Autonomous Driving – a Top-Down-Approach, in: at - Automatisierungstechnik, vol. 63, no. 3, pp. 155–167, 2015

Mayer, W. et al.: Ontologies for Design-Driven Development Processes (2008)
Mayer, W.; Mühlenfeld, A.; Stumptner, M.: Using Ontologies to Optimise Design-Driven Development Processes, in: Collaborative Product and Service Life Cycle Management for a Sustainable World. Advanced Concurrent Engineering, Springer, London, pp. 451–459, 2008

McBride, B. et al.: RDF Schema 1.1 (2014)
McBride, B.; Brickley, D.; Guha, R. V.: RDF Schema 1.1, W3C Recommendation, URL: https://www.w3.org/TR/rdf-schema, 2014, accessed 09.03.2019

McBride, B. et al.: RDF Primer (2004)
McBride, B.; Manola, F.; Miller, E.: RDF Primer, W3C Recommendation, URL: https://www.w3.org/TR/rdf-primer, 2004, accessed 09.03.2019

Menzel, T. et al.: Scenarios for Automated Vehicles (2018)
Menzel, T.; Bagschik, G.; Maurer, M.: Scenarios for Development, Test and Validation of Automated Vehicles, in: IEEE Intelligent Vehicles Symposium, pp. 1821–1827, 2018

Mizoguchi, R.: Ontology Development, Tools and Languages (2004)
Mizoguchi, R.: Tutorial on Ontological Engineering – Part 2: Ontology Development, Tools and Languages, in: New Generation Computing, vol. 22, iss. 1, pp. 61–96, 2004

Moore, R. L.: Human Factors Affecting Design of Vehicles and Roads (1969)
Moore, R. L.: Some Human Factors Affecting the Design of Vehicles and Roads, in: Journal of the Institute of Highway Engineers, vol. 16, pp. 13–22, 1969

National Highway Traffic Safety Administration (NHTSA): Automated Driving – A Vision for Safety (2017)
National Highway Traffic Safety Administration (NHTSA): Automated Driving Systems 2.0 – A Vision for Safety, URL: https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13069a-ads2.0_090617_v9a_tag.pdf, 2017, accessed 14.02.2019

Neches, R. et al.: Enabling Technology for Knowledge Sharing (1991)
Neches, R.; Fikes, R. E.; Finin, T.; Gruber, T. R.; Senator, T.; Swartout, W. R.: Enabling Technology for Knowledge Sharing, in: AI Magazine, vol. 12, no. 3, pp. 36–56, 1991

NetworkX developers: NetworkX – Software for Complex Networks (2018)
NetworkX developers: NetworkX – Software for Complex Networks, URL: https://networkx.github.io/, 2018, accessed 07.04.2019

Nolte, M. et al.: Towards a Skill- and Ability-Based Development Process (2017)
Nolte, M.; Bagschik, G.; Jatzkowski, I.; Stolte, T.; Reschka, A.; Maurer, M.: Towards a Skill- and Ability-Based Development Process for Self-Aware Automated Road Vehicles, in: IEEE International Conference on Intelligent Transportation Systems, 2017

Noy, N. F.; McGuinness, D.: Ontology Development 101 (2001)
Noy, N. F.; McGuinness, D.: Ontology Development 101: A Guide to creating your first Ontology, Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 und Stanford Medical Informatics Technical Report SMI-2001-0880, 2001

Patel-Schneider, P. F. et al.: OWL Web Ontology Language (2004)
Patel-Schneider, P. F.; Hayes, P.; Horrocks, I.: OWL Web Ontology Language Semantics and Abstract Syntax, W3C Recommendation, URL: https://www.w3.org/TR/owl-semantics, 2004, accessed 09.03.2019

Pérez, J. et al.: Semantics and Complexity of SPARQL (2006)
Pérez, J.; Arenas, M.; Gutierrez, C.: Semantics and Complexity of SPARQL, in: Cruz, I. et al. (eds.), The Semantic Web – ISWC 2006. Lecture Notes in Computer Science, vol. 4273, Springer, Berlin, Heidelberg, pp. 30–43, 2006

Prud'hommeaux, E.; Seaborne, A.: SPARQL Query Language for RDF (2008)
Prud'hommeaux, E.; Seaborne, A.: SPARQL Query Language for RDF, W3C Recommendation, URL: https://www.w3.org/TR/rdf-sparql-query, 2008, accessed 09.03.2019

Reschka, A.: Diss., Fertigkeiten- und Fähigkeitengraphen (2017)
Reschka, A.: Fertigkeiten- und Fähigkeitengraphen als Grundlage für den sicheren Betrieb von automatisierten Fahrzeugen in städtischer Umgebung, Dissertation TU Braunschweig, 2017

Reschka, A. et al.: Ability and Skill Graphs (2015)
Reschka, A.; Bagschik, G.; Ulbrich, S.; Nolte, M.; Maurer, M.: Ability and Skill Graphs for System Modeling, Online Monitoring and Decision Support for Vehicle Guidance Systems, in: IEEE Intelligent Vehicles Symposium, 2015

Rossig, W. E.: Wissenschaftliche Arbeiten (2011)
Rossig, W. E.: Wissenschaftliche Arbeiten – Leitfaden für Haus- und Seminararbeiten, Bachelor- und Masterthesis, Diplom- und Magisterarbeiten, Dissertationen, BerlinDruck, Achim, 2011

Rumar, K.: The Human Factor of Road Safety (1982)
Rumar, K.: The Human Factor of Road Safety, in: Australian Road Research Board Proceedings, vol. 11, iss. 1, pp. 63–80, 1982

Schuldt, F.: Diss., Methodisches Testen von automatisierten Fahrfunktionen (2017)
Schuldt, F.: Ein Beitrag für den methodischen Test von automatisierten Fahrfunktionen mit Hilfe von virtuellen Umgebungen, Dissertation TU Braunschweig, 2017

Siegemund, K. et al.: Ontology-Driven Requirements Engineering (2011)
Siegemund, K.; Assmann, U.; Pan, J.; Thomas, E. J.; Zhao, Y.: Towards Ontology-Driven Requirements Engineering, in: International Semantic Web Conference, vol. 10, 2011

Stanford Center for Biomedical Informatics Research: Protégé (2016)
Stanford Center for Biomedical Informatics Research: Protégé, URL: https://protege.stanford.edu, 2016, accessed 23.03.2019

Stanford Center for Biomedical Informatics Research: Protégé Wiki (2016)
Stanford Center for Biomedical Informatics Research: Protégé Wiki, URL: https://protegewiki.stanford.edu/wiki, 2016, accessed 05.04.2019

Steimle, M. et al.: Terminologie für den szenarienbasierten Testansatz (2018)
Steimle, M.; Bagschik, G.; Menzel, T.; Wendler, J. T.; Maurer, M.: Ein Beitrag zur Terminologie für den szenarienbasierten Testansatz automatisierter Fahrfunktionen, in: AAET – Automatisiertes und vernetztes Fahren, 2018

Studer, R. et al.: Knowledge Engineering: Principles and Methods (1998)
Studer, R.; Benjamins, V. R.; Fensel, D.: Knowledge Engineering: Principles and Methods, in: IEEE Transactions on Data and Knowledge Engineering, vol. 25, iss. 1–2, pp. 161–197, 1998

The Rule Markup Initiative: RuleML – Realize your Knowledge (2019)
The Rule Markup Initiative: RuleML – Realize your Knowledge, URL: http://ruleml.org, 2019, accessed 18.03.2019

Thorn, E. et al.: Automated Driving System Testable Cases and Scenarios (2018)
Thorn, E.; Kimmel, S.; Chaka, M.: A Framework for Automated Driving System Testable Cases and Scenarios (Report no. DOT HS 812 623), National Highway Traffic Safety Administration, Washington, DC, 2018

Ulbrich, S. et al.: The Terms Scene, Situation and Scenario (2015)
Ulbrich, S.; Menzel, T.; Reschka, A.; Schuldt, F.; Maurer, M.: Defining and Substantiating the Terms Scene, Situation, and Scenario for Automated Driving, in: IEEE International Conference on Intelligent Transportation Systems, 2015

W3C: World Wide Web Consortium (2019)
W3C: World Wide Web Consortium, URL: https://www.w3.org, 2019, accessed 17.03.2019

Yang, Z. et al.: Evaluation Metrics for Ontology Complexity (2006)
Yang, Z.; Zhang, D.; Ye, C.: Evaluation Metrics for Ontology Complexity and Evolution Analysis, in: IEEE International Conference on e-Business Engineering, pp. 162–170, 2006

Ziegler, J. et al.: Making Bertha Drive (2014)
Ziegler, J.; Dang, T.; Franke, U.; Lategahn, H.; Bender, P.; Schreiber, M.; Strauss, T.; Appenrodt, N.; Keller, C. G.; Kaus, E.; Stiller, C.; Herrtwich, R. G.: Making Bertha Drive – An Autonomous Journey on a Historic Route, in: IEEE Intelligent Transportation Systems Magazine, vol. 6, iss. 2, pp. 8–20, 2014